09:51:30 OCA PAD AMENDMENT - PROJECT HEADER INFORMATION 10/06/93

Active

Project #: C-36-692 Cost share #: Rev #: 2
Center # : 10/24-6-R7599-0A0 Center shr #: OCA file #:
Work type : RES
Contract#: 1491 Mod #: LTR DTD 8/16/93 Document : AGR
Prime #: Contract entity: GTRC

Subprojects ? : N CFDA: N/A
Main project #: PE #: N/A

Project unit: COMPUTING Unit code: 02.010.300
Project director(s):
RAMACHANDRAN U COMPUTING (404)894-5136

Sponsor/division names: DIGITAL EQUIPMENT CORP /
Sponsor/division codes: 203 / 011

Award period: 920430 to 940430 (performance) 940430 (reports)

Sponsor amount	New this change	Total to date
Contract value	0.00	0.00
Funded	0.00	0.00
Cost sharing amount		0.00

Does subcontracting plan apply ?: N

Title: RESEARCH ON MASSIVELY PARALLEL ARCHITECTURES

PROJECT ADMINISTRATION DATA

OCA contact: Robert D. Simpkins 894-4820

Sponsor technical contact

MARCO ANNARATONE
(508)493-2869

MASSIVELY PARALLEL SYSTEMS GROUP
DIGITAL EQUIPMENT CORPORATION
146 MAIN STREET, ML01-5/U46
MAYNARD, MA 01754

Sponsor issuing office

BARBARA AUGUSTA, ADMINISTRATOR
(508)493-5125

EXTERNAL RESEARCH PROGRAM
DIGITAL EQUIPMENT CORPORATION
146 MAIN STREET, ML01-3/B11
MAYNARD, MA 01754

Security class (U,C,S,TS) : U ONR resident rep. is ACO (Y/N): N
Defense priority rating : N/A N/A supplemental sheet
Equipment title vests with: Sponsor GIT X

Administrative comments -
NO COST EXTENSION ISSUED BY LETTER DATED 8/16/93.

SR428

GEORGIA INSTITUTE OF TECHNOLOGY
OFFICE OF CONTRACT ADMINISTRATION

NOTICE OF PROJECT CLOSEOUT

Closeout Notice Date 01/05/95

Project No. C-36-692 Center No. 10/24-6-R7599-0A0

Project Director RAMACHANDRAN U School/Lab COMPUTING

Sponsor DIGITAL EQUIPMENT CORP/

Contract/Grant No. 1491 Contract Entity GTRC

Prime Contract No.

Title RESEARCH ON MASSIVELY PARALLEL ARCHITECTURES

Effective Completion Date 940430 (Performance) 940430 (Reports)

Closeout Actions Required:	Y/N	Date Submitted
Final Invoice or Copy of Final Invoice	Y	
Final Report of Inventions and/or Subcontracts	Y	
Government Property Inventory & Related Certificate	N	
Classified Material Certificate	N	
Release and Assignment	N	
Other	N	

Comments

Subproject Under Main Project No.

Continues Project No.

Distribution Required:

Project Director	Y
Administrative Network Representative	Y
GTRI Accounting/Grants and Contracts	Y
Procurement/Supply Services	Y
Research Property Managment	Y
Research Security Services	N
Reports Coordinator (OCA)	Y
GTRC	Y
Project File	Y
Other	N
	N

NOTE: Final Patent Questionnaire sent to PDPI.

Research on Massively Parallel Architectures

Umakishore Ramachandran
Associate Professor
College of Computing
Georgia Tech
Atlanta, Ga 30332

Annual Report, contract number 1491
Project number C-36-692
Digital Equipment Corporation
September 1993

1 Pointer to Deliverables

The main thrust of our proposed study is the inter-relationship between parallel algorithms and architectures. The first year deliverables are a set of parallel codes on parallel machines. The codes are available on ftp.cc.gatech.edu (130.207.7.245). Anonymous ftp is supported on this machine and the codes may be obtained as follows:

```
------
ftp ftp.cc.gatech.edu
<login as anonymous>
cd pub/architecture/sources
ls
<you will find a README file, and subdirectories>
...
...
------
```

The rest of this report gives a short synopsis of the codes that can be found in that ftp site.

2 Parallel Codes

The ftp directory (pub/architecture/sources) contains the source files of several parallel kernels for a range of different parallel machines.

The directory has a subdirectory for each parallel machine. The machines used in the distribution are :

- *cube* : the Intel Hypercube iPSC/860,
- *maspar* : the MasPar DECmpp12000SX,
- *ksr* : the KSR-1,
- *sgi* : the Silicon Graphics Inc. SGI,
- *pvm* : the Parallel Virtual Machine (PVM) software utility that provides a uniform parallel platform over a wide range of machines connected by a local area network.

Each machine directory has subdirectories each containing the sources for a parallel kernel. The parallel kernels in the distribution are :

- *the NAS Parallel Kernels* : A set of computationally intensive codes that represent a wide range of computational fluid dynamics applications. They include :
  - *ep* : An "Embarassingly Parallel" kernel that needs very little communication between processors.
  - *is* : An "Integer Sort" kernel that uses bucket sort to sort a list of integers.
  - *cg* : A "Conjugate Gradient" kernel for solving sparse linear systems.
  - *ft* : A "Fast Fourier Transform" kernel for a 3-D space of complex double precision numbers.
- *the Image Understanding Kernels* : Kernels taken from the Darpa Image Understanding Benchmark used in transforming low pixel-level information to a much higher object-level representation.
  - *filter* : A "Filter" kernel that uses near-neighbor averaging to sharpen an image.
  - *label* : A "Region Labeling" kernel that assigns a unique label to each region in the image.
  - *corner* : A "K-curvature" kernel that identifies the corners in a region.
- *the SPLASH Application Suite* : Applications available from Stanford University for public distribution that are meant to be used in evaluating shared memory multiprocessors. The applications selected from this suite are :
  - *cholesky* : A "Cholesky Factorization" kernel for sparse matrices.
  - *mp3d* : A "Fluid Flow Simulation" kernel that uses Monte Carlo methods to simulate the trajectories of molecules.

The kernels available for each machine in this distribution are given below :
cube : ep, is, cg, ft
maspar : ep, is, filter, label, corner
pvm : ep, is, cg, ft
ksr : ep, is, cg, ft, filter, label, corner, cholesky, mp3d
sgi : mp3d

OFFICE OF CONTRACT ADMINISTRATION
DELIVERABLE SCHEDULE

PROJECT NO. C-36-692 SPONSOR/DIVISION: DIGITAL EQUIPMENT CORP/ 203/011 CONTRACT THRU: GTRC
AWARD DOCUMENT: AGR CONTRACT NUMBER: 1491 MOD NO.: ADMINISTRATIVE
PROJECT.DIRECTOR(S)
RAMACHANDRAN U

UNIT:02.010.300 COMPUTING
INITIATION DATE 920430 TERMINATION PERF DATE 930430 TERMINATION RPTS DATE 930430
PROJECT TITLE: RESEARCH ON MASSIVELY PARALLEL ARCHITECTURES PD/PI CONCUR DATE:_____

REV NO.	DESCRIPTION OF DELIVERABLE	DELIV NO.	PERIOD COVERED (1)	DUE DATE TO SPONSOR (1)	COPIES REQD	REFERENCES	DATE MAILED (2)
1	SET OF PARALLEL KERNELS	1		930430	1	_____	

PLEASE NOTE:
1. Blanks in fields "Period Covered" or "Due Date to Sponsor", indicate "as appropriate" or "as required."
2. Blanks in "Date Mailed" indicate that neither delivery nor notification of delivery has been accomplished through OCA/CSD.

Prepare reports in accordance with:
AGREEMENT EXHIBIT A - DELIVERABLES

Research on Massively Parallel Architectures

Umakishore Ramachandran
Associate Professor
College of Computing
Georgia Tech
Atlanta, Ga 30332

Final Report, contract number 1491
Project number C-36-692
Digital Equipment Corporation
Technical Sponsor: Dr. Marco Annaratone, DEC-HPC Group
December 1994

1 Introduction

This is the final report for the above mentioned project. The main thrust of the study undertaken in this project is the inter-relationship between parallel algorithms and architectures. The approach was a combination of experimentation on real parallel machines such as DECmpp 12000 (aka MasPar MP-2), and KSR-1/2; and simulation to get a better handle on the overheads in a parallel system. The deliverables from this project are a set of parallel codes for a variety of parallel machines; and results of scalability studies of parallel systems. In Section 2 we give a synopsis of the codes that have been developed under this project. Section 3 gives a summary of research accomplishments facilitated by this grant and a bibliography of publications that acknowledge this grant. The publications and the source codes are all available by anonymous ftp from the ftp.cc.gatech.edu, as well as through the World Wide Web (WWW) at URL http://www.cc.gatech.edu/computing/Architecture.

The papers can be found in the directories pub/groups/architecture/TASS, and pub/groups/architecture/Beehive on the ftp site, and the source codes can be found in pub/groups/architecture/sources on the same ftp site. A selection of these papers are enclosed with this final report.

We have developed a simulator for scalability studies of parallel systems called SPASM which takes as input a parallel architecture specification, and uses a suite of applications as the workload for evaluating the parallel architecture. This simulator is also available for distribution and enquiries regarding this can be addressed to Professor Ramachandran (e-mail: rama@cc.gatech.edu).

2 Parallel Codes

The ftp directory (pub/groups/architecture/sources) contains the source files of several parallel kernels for a range of different parallel machines.

The directory has a subdirectory for each parallel machine. The machines used in the distribution are :

- *cube* : the Intel Hypercube iPSC/860,
- *maspar* : the MasPar DECmpp12000SX,
- *ksr* : the KSR-1,
- *sgi* : the Silicon Graphics Inc. SGI,
- *pvm* : the Parallel Virtual Machine (PVM) software utility that provides a uniform parallel platform over a wide range of machines connected by a local area network.
- *SPASM*: this is a home-grown parallel architecture simulator that can model any parallel architecture.

Each machine directory has subdirectories each containing the sources for a parallel kernel. The parallel kernels in the distribution are :

- *the NAS Parallel Kernels* : A set of computationally intensive codes that represent a wide range of computational fluid dynamics applications. They include :
  - *ep* : An "Embarassingly Parallel" kernel that needs very little communication between processors.
  - *is* : An "Integer Sort" kernel that uses bucket sort to sort a list of integers.
  - *cg* : A "Conjugate Gradient" kernel for solving sparse linear systems.
  - *ft* : A "Fast Fourier Transform" kernel for a 3-D space of complex double precision numbers.
- *the Image Understanding Kernels* : Kernels taken from the Darpa Image Understanding Benchmark used in transforming low pixel-level information to a much higher object-level representation.
  - *filter* : A "Filter" kernel that uses near-neighbor averaging to sharpen an image.
  - *label* : A "Region Labeling" kernel that assigns a unique label to each region in the image.
  - *corner* : A "K-curvature" kernel that identifies the corners in a region.
- *the SPLASH Application Suite* : Applications available from Stanford University for public distribution that are meant to be used in evaluating shared memory multiprocessors. The applications selected from this suite are :
  - *cholesky* : A "Cholesky Factorization" kernel for sparse matrices.
  - *mp3d* : A "Fluid Flow Simulation" kernel that uses Monte Carlo methods to simulate the trajectories of molecules.
  - *Barnes-Hut*: An N-body simulation program.
- *Combinatorial Optimization Codes*:
  - *GAP*: The Generalized assignment problem (GAP) asks for a maximum profit assignment of n tasks to m agents such that each task is assigned to precisely one agent subject to resource restrictions on the agents.
  - *TSP*: This is the well-known traveling salesman problem.

The kernels available for each machine in this distribution are given below :
cube : ep, is, cg, ft
maspar : ep, is, filter, label, corner
pvm : ep, is, cg, ft, GAP, TSP
ksr : ep, is, cg, ft, filter, label, corner, cholesky, mp3d, GAP, TSP
sgi : mp3d
spasm: ep, is, cg, ft, cholesky, Barnes-Hut.

3 Research Accomplishments

There are two projects underway that are facilitated by this grant. **Beehive** is a project that investigates the software and hardware issues in the design of scalable shared memory multiprocessors. The architectural mechanisms provided by Beehive allow any desired memory model to be supported in a cache-based multiprocessor environment. The key idea is to have the minimal mechanisms in hardware for achieving coherence and leave it up to the compiler and language runtime to trigger the coherence actions when needed based on program semantics, as well as enforcing the desired memory model. We have developed marking algorithms for incorporation in a compiler that maps the loads and stores of the application to the Beehive primitives. Work until now has specified the architectural primitives [1, 2, 3, 4], developed compiler marking algorithms for triggering consistency actions [5], and carried out preliminary quantitative evaluation that clearly shows the performance advantage of this approach over the conventional approach of leaving the mechanisms and policies for coherence maintenance entirely in the hardware. Use of optical interconnect technology for efficiently implementing these architectural mechanisms have also been explored [6]. Current work includes data flow analysis algorithms for consistency maintenance and concurrency management, implementation of these algorithms in a compiler/runtime, and performance evaluation through simulation. Parallel implementation of the simulator itself for speeding up the simulation as well as for simulating larger systems (both machine size as well as problem size) is also being investigated [7].

Related to the Beehive project is **TASS** (Top-down Approach to Scalability Studies), the main thrust of which is to study the inter-relationship between parallel applications and architectures [8, 9, 10, 11, 12, 13, 14]. The overheads in a parallel system (an application-architecture combination) that limit its scalability have to be identified and separated in order to enable performance-conscious parallel application design and the development of high-performance parallel machines. The TASS project develops a framework to enable such a study. In this framework (see Figure 1), experimentation (on state-of-the-art parallel machines such as KSR-2) is used in conjunction with simulation to understand the performance of real applications on real architectures, and to identify the interesting kernels that occur in these applications for subsequent use in the simulation studies. The datapoints obtained from simulation are used to derive analytical models as well as refine existing models for predicting the scalability of larger systems. At the heart of this framework is an execution-driven simulation testbed called SPASM which uses a suite of applications as the workload. SPASM identifies and separates all the overheads in a parallel system that are of interest from a performance standpoint. Current and ongoing work include using this framework for identifying algorithmic and architectural bottlenecks in a parallel system, predicting the performance of an application on a larger configuration of an existing architecture, studying the scalability of a specific architecture with respect to a suite of applications, and selecting the best architecture platform for an application domain. Recent results using this framework include understanding the performance characteristics of scientific applications on message-passing and shared memory platforms [15, 16] with different communication topologies; illustrating the use of machine abstractions for performance studies of parallel systems [17]; synthesizing network requirements for parallel scientific applications [18, 19];

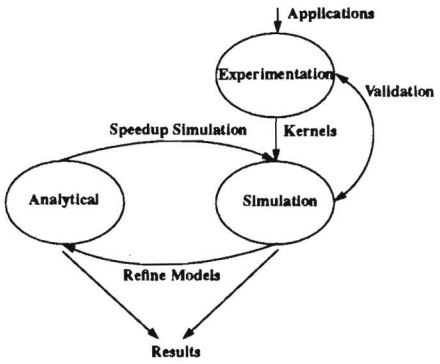

Figure 1: A Framework for Scalability Study

and synthesizing architectural mechanisms for explicit communication in shared memory multiprocessors [20].

Three students, jointly funded by the NSF PYI award and this matching equipment grant from DEC, have completed their doctoral dissertations and have taken up academic and industry positions. Dr. Joonwon Lee (graduation date June 1991, thesis title "Architectural Primitives for Scalable Shared Memory Multiprocessors") is an assistant Professor with the Dept. of Information and Communication, Kaist University, South Korea. Dr. Walter B. Ligon III (graduation date August 1992, thesis title, "An Empirical Evaluation of Architectural Reconfigurability") is an assistant Professor with the Department of Electrical and Computer Engineering, Clemson University. Dr. Martin Davis Jr. (graduation date December 1992, thesis title, "Optical Waveguides in General Purpose Parallel Computers") is with SYSTRAN Corp., Dayton, Ohio. Currently, there are four graduate students who are working on aspects related to the above two projects for their doctoral dissertations. Two of my students, Anand Sivasubramaniam and Aman Singla, also contributed to the HPC efforts at DEC through internships. Anand worked with Dr. Marco Annaratone in the MPSG group for 6 months, while Aman Singla worked with Dr. Rishiyur Nikhil and Dr. Bert Halstead at CRL for 3 months. Publications facilitated by the support from the DEC matching award so far total 3 doctoral dissertations, 2 book chapters, 4 journal publications, and 7 peer-reviewed conference presentations. In addition, the PI (Umakishore Ramachandran) developed a tutorial entitled **Multigranular Computing**. The theme of this tutorial was to illustrate the parallelism continuum that exists in fine-grained, medium-grained, and coarse-grained machines. This tutorial covered the granularity spectrum – from fine-grained to coarse-grained – of parallel computation. This has been presented in several leading conferences such as *20th Annual International Symposium on Computer Architecture*, and *1994 SIGMETRICS conference on measurement and modeling of computer systems.*

4 Pointer to Deliverables

As we mentioned all the source codes and papers are available on ftp.cc.gatech.edu. Anonymous ftp is supported on this machine and the codes and papers may be obtained as follows:

```
------
ftp ftp.cc.gatech.edu
<login as anonymous>
cd pub/groups/architecture
ls
<you will find a README file, and subdirectories>
...
...
------
```

References

[1] J. Lee and U. Ramachandran. Synchronization with Multiprocessor Caches. In **Proceedings of the 17th Annual International Symposium on Computer Architecture**, pages 27–37, 1990.

[2] J. Lee, and U. Ramachandran. Architectural Primitives for a Scalable Shared Memory Multiprocessor. In **Proceedings of the Third Annual ACM Symposium on Parallel Algorithms and Architectures**, pages 103–114, Hilton Head, South Carolina, July 1991.

[3] J. Lee, and U. Ramachandran. Locks, Directories, and Weak Coherence – A Recipe for Scalable Shared Memory Multiprocessors. In Dubois and Thakkar (Editors), **Scalable Shared Memory Multiprocessors**, Kluwer Academic Publishers, 1991.

[4] U. Ramachandran, and J. Lee. Cache-based Synchronization in Shared Memory Multiprocessors. **Journal of Parallel and Distributed Computing**. Accepted conditionally subject to final review.

[5] G. Shah, and U. Ramachandran. Towards Exploiting the Architectural Features of Beehive. Technical Report GIT-CC-91/51, College of Computing, Georgia Institute of Technology, November 1991. Appeared in **1993 ISCA workshop on Scalable Shared Memory Multiprocessors.**

[6] M. H. Davis, Jr. Using Optical Waveguides in General Purpose Parallel Computing. **Optical Computing and Processing**, 3(2):103–117, 1993.

[7] Gautam Shah, Umakishore Ramachandran, and Richard Fujimoto. Timepatch: A novel technique for the parallel simulation of multiprocessor caches. Technical Report TR-94-52, Georgia Institute of Technology, October 1994. Submitted for Publication.

[8] U. Ramachandran, and H. Venkateswaran. Towards Realizing Scalable High Performance Parallel Systems. In U. Vishkin, editor, **Developing a Computer Science Agenda for High-Performance Computing**, ACM Press, pages 124-128, 1994.

[9] A. Sivasubramaniam, U. Ramachandran, and H. Venkateswaran. Message-Passing: Computational Model, Programming Paradigm, and Experimental Studies. Technical Report GIT-CC-91/11, College of Computing, Georgia Institute of Technology, February 1991. A short version of this paper appeared in **Proceedings of the Sixth International Parallel Processing Symposium**, March 1992.

[10] A. Sivasubramaniam, G. Shah, J. Lee, U. Ramachandran, and H. Venkateswaran. Experimental Evaluation of Algorithmic Performance on Two Shared Memory Multiprocessors. In Norihisa Suzuki, editor, **Shared Memory Multiprocessing**, pages 81–107. MIT Press, 1992. Also in **Proceedings of the First International Symposium on Shared Memory Multiprocessing**, April 1991.

[11] W. B. Ligon III, and U. Ramachandran. An Empirical Methodology for Exploring Reconfigurable Architectures. **Journal of Parallel and Distributed Computing**, (19):323–337, 1993.

[12] W. B. Ligon III, and U. Ramachandran. Evaluating Multiguage Architectures for Computer Vision. **Journal of Parallel and Distributed Computing**, (21):323–333, 1994. Special issue on Heterogeneous Computing.

[13] U. Ramachandran, G. Shah, S. Ravikumar, and J. Muthukumarasamy. Scalability study of the KSR-1. In **Proceedings of the 1993 International Conference on Parallel Processing**, pages I–237–240, August 1993.

[14] A. Sivasubramaniam, U. Ramachandran, and H. Venkateswaran. A comparative evaluation of techniques for studying parallel system performance. Technical Report GIT-CC-94/38, College of Computing, Georgia Institute of Technology, September 1994. *Submitted for publication.*

[15] A. Sivasubramaniam, A. Singla, U. Ramachandran, and H. Venkateswaran. A Simulation-based Scalability Study of Parallel Systems. **Journal of Parallel and Distributed Computing**, 22(3):411–426, September 1994. Special issue on Scalability of Parallel Algorithms and Architectures.

[16] A. Sivasubramaniam, A. Singla, U. Ramachandran, and H. Venkateswaran. An Approach to Scalability Study of Shared Memory Parallel Systems. In **Proceedings of the ACM SIGMETRICS 1994 Conference on Measurement and Modeling of Computer Systems**, pages 171–180, May 1994.

[17] A. Sivasubramaniam, A. Singla, U. Ramachandran, and H. Venkateswaran. Abstracting network characteristics and locality properties of parallel systems. In **Proceedings of the First International Symposium on High Performance Computer Architecture**, January 1995. To appear.

[18] A. Sivasubramaniam, A. Singla, U. Ramachandran, and H. Venkateswaran. On characterizing bandwidth requirements of parallel applications. Technical Report GIT-CC-94/31, College of Computing, Georgia Institute of Technology, July 1994. *Submitted for publication.*

[19] U. Ramachandran, H. Venkateswaran, A. Sivasubramaniam, and A. Singla. Issues in Understanding the Scalability of Parallel Systems. In **Proceedings of the First International Workshop on Parallel Processing**, December 1994. To appear.

[20] U. Ramachandran, G. Shah, A. Sivasubramaniam, A. Singla, and I. Yanasak. Architectural mechanisms for explicit communication in shared memory multiprocessors. Technical Report GIT-CC-94/59, College of Computing, Georgia Institute of Technology, December 1994. *Submitted for publication.*

In *Proceedings of the ACM SIGMETRICS Conference on Measurement and Modeling of Computer Systems*, pages 171-180, May 1994.

An Approach to Scalability Study of Shared Memory Parallel Systems*

Anand Sivasubramaniam *Aman Singla* *Umakishore Ramachandran* *H. Venkateswaran*

College of Computing
Georgia Institute of Technology
Atlanta, GA 30332-0280.
{*anand, aman, rama, venkat*}*@cc.gatech.edu*

Abstract

The overheads in a parallel system that limit its scalability need to be identified and separated in order to enable parallel algorithm design and the development of parallel machines. Such overheads may be broadly classified into two components. The first one is intrinsic to the algorithm and arises due to factors such as the work-imbalance and the serial fraction. The second one is due to the interaction between the algorithm and the architecture and arises due to latency and contention in the network. A top-down approach to scalability study of shared memory parallel systems is proposed in this research. We define the notion of *overhead functions* associated with the different algorithmic and architectural characteristics to *quantify* the scalability of parallel systems; we isolate the algorithmic overhead and the overheads due to network latency and contention from the overall execution time of an application; we design and implement an execution-driven simulation platform that incorporates these methods for quantifying the overhead functions; and we use this simulator to study the scalability characteristics of five applications on shared memory platforms with different communication topologies.

1 Introduction

Scalability is a notion frequently used to signify the "goodness" of parallel systems, where the term parallel system is used to denote an application-architecture combination. A good understanding of this notion may be used to: select the best architecture platform for an application domain, predict the performance of an application on a larger configuration of an existing architecture, identify application and architectural bottlenecks in a parallel system, and glean insight on the interaction between an application and an architecture to understand the scalability of other application-architecture pairs. In this paper, we develop a framework for studying the inter-play between applications and architectures to understand their implications on scalability. Since real-life applications set the standards for computing, it is meaningful to use such applications for studying the scalability of parallel systems. We call such an application-driven approach a *top-down approach to scalability study*. The main thrust of this approach is to identify the important algorithmic and architectural artifacts that impact the performance of a parallel system, understand the interaction between them, quantify the impact of these artifacts on the execution time of an application, and use these quantifications in studying the scalability of the system.

The main contributions of our work can be summarized as follows: we define the notion of *overhead functions* associated with the different algorithmic and architectural characteristics; we develop a method for separating the algorithmic overhead; we also isolate the overheads due to network *latency* (the actual hardware transmission time in the network) and *contention* (the amount of time spent waiting for a resource to become free in the network) from the overall execution time of an application; we design and implement a simulation platform that quantifies these overheads; and we use this simulator to study the scalability of five applications on shared memory platforms with three different network topologies.

Performance metrics such as speedup [2], scaled speedup [11], sizeup [25], experimentally determined serial fraction [12], and isoefficiency function [13] have been proposed for quantifying the scalability of parallel systems. While these metrics are extremely useful for tracking performance trends, they do not provide adequate information needed to understand the reason why an application does not scale well on an architecture. The overhead functions that we identify, separate, and quantify in this work, help us overcome this inadequacy. We are not aware of any other work that separates these overheads (in the context of real applications), and believe that such a separation is very important in understanding the interaction between applications and architectures. The growth of overhead functions will provide key insights on the scalability of a parallel system by suggesting application restructuring, as well as architectural enhancements.

Several performance studies address issues such as latency, contention and synchronization. The scalability of synchronization primitives supported by the hardware [3, 15] and the limits on interconnection network performance [1, 16] are examples of such studies. While such issues are extremely important, it is necessary to put the impact of these factors into perspective by considering them in the context of overall application performance. There are studies that use real applications to address specific issues like the effect of sharing in parallel programs on the cache and bus performance [10] and the impact of synchronization and task granularity on parallel system performance [6]. Cypher et al. [9] identify the architectural requirements such as floating point operations, communication, and input/output for message-passing scientific applications. Rothberg et al. [18] conduct a similar study towards identifying the cache and memory size requirements for several applications. However, there have been very few attempts at quantifying the effects of algorithmic and architectural interactions in a parallel system.

This work is part of a larger project which aims at understanding

*This work has been funded in part by NSF grants MIPS-9058430 and MIPS-9200005, and an equipment grant from DEC.

the significant issues in the design of scalable parallel systems using the above-mentioned top-down approach. In our earlier work, we studied issues such as task granularity, data distribution, scheduling, and synchronization, by implementing frequently used parallel algorithms on shared memory [21] and message-passing [20] platforms. In [24], we illustrated the top-down approach for the scalability study of message-passing systems. In this paper, we conduct a similar study for shared memory systems. In a companion paper [23] we evaluate the use of abstractions for the network and locality in the context of simulating cache-coherent shared memory multiprocessors.

The top-down approach and the overhead functions are elaborated in Section 2. Details of our simulation platform, SPASM (Simulator for Parallel Architectural Scalability Measurements), which quantifies these overhead functions are also discussed in this section. The characteristics of the five applications used in this study are summarized in Section 3, details of the three shared memory platforms are presented in Section 4, and the results of our study with their implications on scalability are summarized in Section 5. Concluding remarks are presented in Section 6.

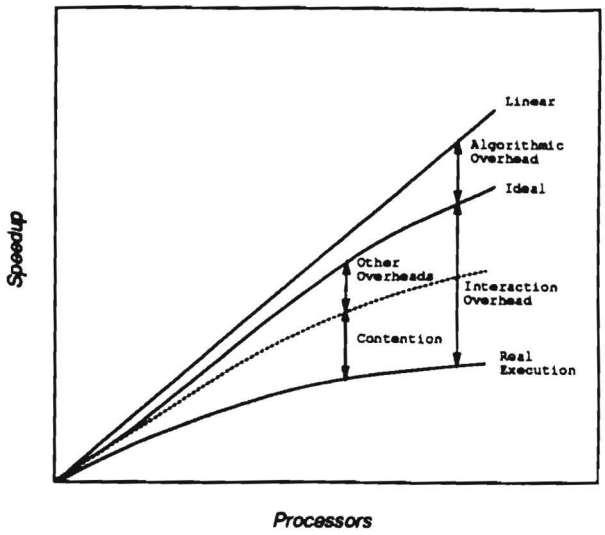

Figure 1: Top-down Approach to Scalability Study

2 Top-Down Approach

Adhering to the RISC ideology in the evolution of sequential architectures, we would like to use *real world applications* in the performance evaluation of parallel machines. However, applications normally tend to contain large volumes of code that are not easily portable and a level of detail that is not very familiar to someone outside that application domain. Hence, computer scientists have traditionally used parallel algorithms that capture the interesting computation phases of applications for benchmarking their machines. Such abstractions of real applications that capture the main phases of the computation are called *kernels*. One can go even lower than kernels by abstracting the main *loops* in the computation (like the Lawrence Livermore loops [14]) and evaluating their performance. As one goes lower, the outcome of the evaluation becomes less realistic. Even though an application may be abstracted by the kernels inside it, the sum of the times spent in the underlying kernels may not necessarily yield the time taken by the application. There is usually a cost involved in moving from one kernel to another such as the data movements and rearrangements in an application that are not part of the kernels that it is comprised of. For instance, an efficient implementation of a kernel may need to have the input data organized in a certain fashion which may not necessarily be the format of the output from the preceding kernel in the application. Despite its limitations, we believe that the scalability of an application with respect to an architecture can be captured by studying its kernels, since they represent the computationally intensive phases of an application. Therefore, we have used kernels in this study.

Parallel system overheads (see Figure 1) may be broadly classified into a purely algorithmic component (*algorithmic overhead*), and a component arising from the interaction of the algorithm and the architecture (*interaction overhead*). The algorithmic overhead is quantified by computing the time taken for execution of a given parallel program on an ideal machine such as the PRAM [26] and measuring its deviation from a linear speedup curve. A real execution could deviate significantly from the ideal execution due to overheads such as latency, contention, synchronization, scheduling and cache effects. These overheads are lumped together as the interaction overhead. In an architecture with no contention overhead, the communication pattern of the application would dictate the latency overhead incurred by it. Thus the performance of an application (on an architecture devoid of network contention) may lie between the ideal curve and the real execution curve (see Figure 1). Therefore, to fully understand the scalability of a parallel system it is important to isolate the influence of each component of the interaction overhead on the overall performance.

The key elements of our top-down approach for studying the scalability of parallel systems are:

- experiment with real world applications
- identify parallel kernels that occur in these applications
- study the interaction of these kernels with architectural features to separate and quantify the overheads in the parallel system
- use these overheads for predicting the scalability of parallel systems.

2.1 Implementing the Top-Down Approach

Scalability study of parallel systems is complex due to the several degrees of freedom that they exhibit. Experimentation, simulation, and analytical models are three techniques that have been commonly used in such studies. But it is well-known that each has its own limitations. The main focus of our top-down approach is to quantify the overheads that arise from the interaction between the kernels and the architecture and their impact on the overall execution of the application. Experimentation on real architectures does not allow studying the effects of changing individual architectural parameters on the performance. It is not clear that analytical models can realistically capture the complex and dynamic interactions between applications and architectures. Therefore, we use simulation for quantifying and separating the overheads.

Our simulation platform (SPASM), to be presented in the next sub-section, provides an elegant set of mechanisms for quantifying the different overheads we discussed earlier. The algorithmic overhead is quantified by computing the time taken for execution of a given parallel program on an ideal machine such as the PRAM [26] and measuring its deviation from a linear speedup curve. The interaction overhead is also separated into its component parts. We currently do not address scheduling overheads[1]. Accesses to variables in a shared memory system may involve the network, and the

[1]We do not distinguish between the terms, *process*, *processor* and *thread*, and use them synonymously in this paper.

physical limitations of the network tend to contribute to overheads in the execution. These overheads may be broadly classified as latency and contention, and we associate an overhead function with each. The *Latency Overhead Function* is thus defined as the total amount of time spent by a processor waiting for messages due to the transmission time on the links and the switching overhead in the network assuming that the messages did not have to contend for any link. Likewise, the *Contention Overhead Function* is the total amount of time incurred by a processor due to the time spent waiting for links to become free by the messages. Shared memory systems normally provide some synchronization support that is as simple as an atomic read-modify-write operation, or may provide special hardware for more complicated operations like barriers and queue-based locks. While the latter may save execution time for complicated synchronization operations, the former is more flexible for implementing a variety of such operations. For reasons of generality, we assume that only the test&set operation is supported by shared memory systems. We also assume that the memory module (at which the operation is performed), is intelligent enough to perform the necessary operation in unit time. With such an assumption, the only network overhead due to the synchronization operation (test&set) is a roundtrip message, and the overheads for such a message are accounted for in the latency and contention overhead functions described earlier. The waiting time incurred by a processor during synchronization operations is accounted for in the CPU time which would manifest itself as an algorithmic overhead. The statistics (CPU time, latency overhead, and contention overhead) are quantified and presented for each interesting mode of the program execution (see Section 2.2).

Constant problem size (where the problem size remains unchanged as the number of processors is increased), *memory constrained* (where the problem size is scaled up linearly with the number of processors), and *time constrained* (where the problem size is scaled up to keep the execution time constant with increasing number of processors) are three well-accepted scaling models used in the study of parallel systems. Overhead functions can be used to study the growth of system overheads for any of these scaling strategies. In our simulation experiments, we limit ourselves to the constant problem size scaling model.

2.2 SPASM

SPASM is an execution-driven simulator written in CSIM. As with other recent simulators [5, 7, 17], the bulk of the instructions in the parallel program is executed at the speed of the native processor (SPARC in this study) and only the instructions (such as LOADS and STORES) that may potentially involve a network access are simulated. The input to the simulator are parallel applications written in C. These programs are pre-processed (to label shared memory accesses), the compiled assembly code is augmented with cycle counting instructions, and the assembled binary is linked with the simulator code. The system parameters that can be specified to SPASM are: the *number of processors (p)*, the *clock speed* of the processor, the *hardware bandwidth* of the links in the network, and the *switching delays*.

2.2.1 Metrics

SPASM provides a wide range of statistical information about the execution of the program. It gives the *total time* (simulated time) which is the maximum of the running times of the individual parallel processors. This is the time that would be taken by an execution of the parallel program on the target parallel machine. *Speedup* using p processors is measured as the ratio of the total time on 1 processor to the total time on p processors.

Ideal time is the total time taken by a parallel program to execute on an ideal machine such as the PRAM. It includes the algorithmic overhead but does not include the interaction overhead. SPASM simulates an ideal machine to provide this metric. As we mentioned in Section 2, the difference between the linear time and the ideal time gives the algorithmic overhead.

SPASM quantifies both the latency overhead function as well as the contention overhead function seen by a processor as described in Section 2. This is done by time-stamping messages when they are sent. At the time a message is received, the time that the message would have taken in a contention free environment is charged to the latency overhead function while the rest of the time is accounted for in the contention overhead function. Though not relevant to this study, it is worthwhile to mention that SPASM provides the latency and contention incurred by a message as well as the latency and contention that a processor may choose to see. Even though a message may incur a certain latency and contention, a processor may choose to hide all or part of it by overlapping computation with communication. Such a scenario may arise with a non-blocking message operation on a message-passing machine or with a prefetch operation on a shared memory machine. But for the rest of this paper (since we deal with blocking load/store shared memory operations), we assume that a processor sees all of the network latency and contention.

SPASM also provides statistical information about the network. It gives the utilization of each link in the network and the average queue lengths of messages at any particular link. This information can be useful in identifying network bottlenecks and comparing relative merits of different networks and their capabilities.

It is often useful to have the above metrics for different modes of execution of the algorithm. Such a breakup would help identify bottlenecks in the program, and also help estimate the potential gain in performance that may be possible through a specific hardware or software enhancement. SPASM provides statistics grouped together for system-defined as well as for user-defined modes of execution. The system-defined modes are:

- NORMAL: A program is in the NORMAL mode if it is not in any of the other modes. An application programmer may further define sub-modes if necessary.
- BARRIER: Mode corresponding to a barrier synchronization operation.
- MUTEX: Even though the simulated hardware provides only a test&set operation, mutual exclusion *lock* (implemented using test-test&set [3]) is available as a library function in SPASM. A program enters this mode during lock operations. With this mechanism, we can separate the overheads due to the synchronization operations from the rest of the program execution.
- PGM_SYNC: Parallel programs may use Signal-Wait semantics for pairwise synchronization. A lock is unnecessary for the Signal variable since only 1 processor writes into it and the other reads from it. This mode is used to differentiate such accesses from normal load/store accesses.

The *total time* for a given application is the sum of the *execution times* for each of the above defined modes. The *execution time* for each program mode is the sum of the *computation time*, the *latency overhead* and the *contention overhead* observed in the mode. The metrics identified by SPASM quantify the algorithmic overhead and the interesting components of the interaction overhead. Computation time in the NORMAL mode is the actual time spent in local computation in an application. The sum of latency and contention overheads in the NORMAL mode is the actual time incurred for ordinary data accesses. For the BARRIER and PGM_SYNC modes,

the computation time is the wait time incurred by a processor in synchronizing with other processors that results from the algorithmic work imbalance. The computation time in the MUTEX mode is the time spent in waiting for a lock and represents the serial part in an application arising due to critical sections. For the BARRIER and MUTEX modes, the computation time also includes the cost of implementing the synchronization primitive and other residual effects due to latency and contention for prior accesses. In all three synchronization modes, the latency and contention overheads together represent the actual time incurred in accessing synchronization variables.

3 Application Characteristics

Three of the applications (EP, IS and CG) are from the NAS parallel benchmark suite [4]; CHOLESKY is from the SPLASH benchmark suite [19]; and FFT is the well-known Fast Fourier Transform algorithm. EP and FFT are well-structured applications with regular communication patterns determinable at compile-time, with the difference that EP has a higher computation to communication ratio. IS also has a regular communication pattern, but in addition it uses locks for mutual exclusion during the execution. CG and CHOLESKY are different from the other applications in that their communication patterns are not regular (both use sparse matrices) and cannot be determined at compile time. While a certain number of rows of the matrix in CG is assigned to a processor at compile time (static scheduling), CHOLESKY uses a dynamically maintained queue of runnable tasks. The reader is referred to [22] for further details of the applications.

4 Architectural Characteristics

Since uniprocessor architecture is getting standardized with the advent of RISC technology, we fix most of the processor characteristics by using a 33 MHz SPARC chip as the baseline for each processor in a parallel system. Such an assumption enables us to make a fair comparison of the relative merits of the interesting parallel architectural characteristics across different platforms. Input-output characteristics are beyond the purview of this study.

We use three shared memory platforms with different interconnection topologies: the *fully connected network*, the *binary hypercube* and the *2-D mesh*. All three networks use serial (1-bit wide) unidirectional links with a link bandwidth of 20 MBytes/sec. The fully connected network models two links (one in each direction) between every pair of processors in the system. The cube platform connects the processors in a bidirectional binary hypercube topology and uses the *e*-cube algorithm for routing. The 2-D mesh resembles the Intel Touchstone Delta system. Links in the North, South, East and West directions, enable a processor in the middle of the mesh to communicate with its four immediate neighbors. Processors at corners and along an edge have only two and three neighbors respectively. Equal number of rows and columns is assumed when the number of processors is an even power of 2. Otherwise, the number of columns is twice the number of rows (we restrict the number of processors to a power of 2 in this study). Messages in the mesh are routed along the row until they reach the destination column, upon which they are routed along the column. Messages on all three platforms are circuit-switched using a wormhole routing strategy and the switching delay is assumed to be negligible.

The simulated shared memory hierarchy is CC-NUMA (Cache Coherent Non-Uniform Memory Access). Each node in the system has a sufficiently large piece of the globally shared memory such that for the applications considered, the data-set assigned to each processor fits entirely in its portion of shared memory. There is also a 2-way set-associative private cache (64KBytes with 32 byte blocks) at each node that is maintained sequentially consistent using an invalidation-based fully-mapped directory-based cache coherence scheme. The memory access time is assumed to be 5 CPU cycles, while the cache access time is assumed to be 1 CPU cycle.

5 Performance Results

In this section, we present results from our simulation experiments showing the growth of the overhead functions with respect to the number of processors and their impact on scalability. The simulator allows one to explore the effect of varying other system parameters such as link bandwidth and processor speed on scalability. Since the main focus of this paper is an approach to scalability study, we have not dwelled on the scalability of parallel systems with respect to specific architectural artifacts to any great extent in this paper. We also briefly describe the impact of problem sizes on the system scalability for each kernel.

Figures 2, 3, 4, 5 and 6 show the "ideal" speedup curves (section 2) for the kernels EP, IS, FFT, CG and CHOLESKY, as well as the speedup curves for these kernels on the three hardware platforms. There is negligible deviation from the ideal curve for the EP kernel on the three hardware platforms; a marginal difference for FFT and CG; and a significant deviation for IS and CHOLESKY. For each of these kernels, we quantify the different interaction overheads responsible for the deviation during each execution mode of the kernel. Only the results for IS, FFT and CHOLESKY are discussed in this section due to space constraints. Details on the other kernels can be found in [22].

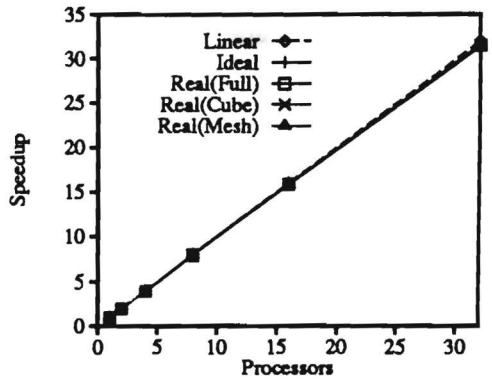

Figure 2: EP: Speedup

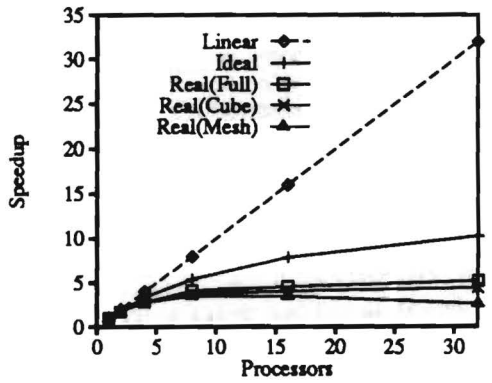

Figure 3: IS: Speedup

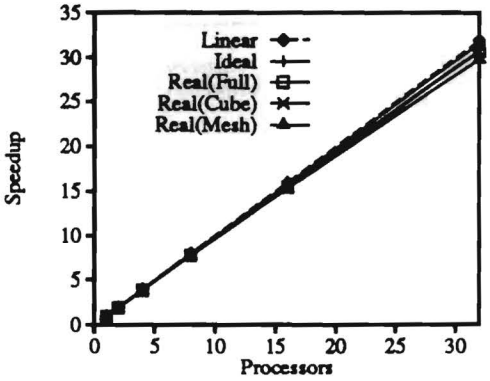

Figure 4: FFT: Speedup

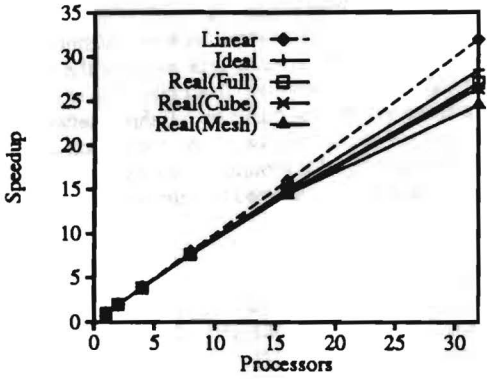

Figure 5: CG: Speedup

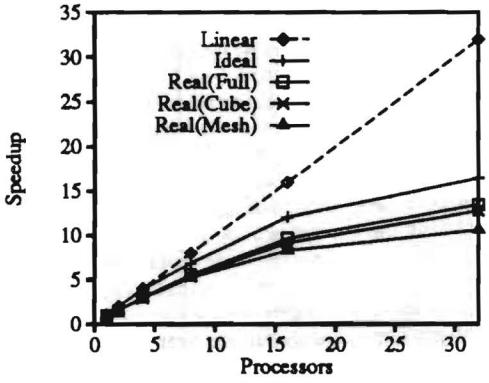

Figure 6: CHOLESKY: Speedup

In the following subsections, we show for each kernel the execution time, the latency, and the contention overhead graphs for the mesh platform. The first shows the total execution time, while the latter two show the communication overheads ignoring the computation time. In each of these graphs, we show the curves for the individual modes of execution applicable for a particular kernel. We also present for each kernel the latency and contention overhead curves on the three architecture platforms. The latency overhead in the NORMAL mode (i.e. due to ordinary data access) is determined by the memory reference pattern of the kernel and the network traffic due to cache line replacement. With sufficiently large size cache at each node, it is reasonable to assume that this latency overhead is only due to the kernel, and thus is expected to be independent of the network topology. Due to the vagaries of the synchronization accesses, it is conceivable that the corresponding latency overheads could differ across network platforms for the other modes. However, in our experiments we have not seen any significant deviation. As a result, the latency overhead curves for all the kernels look alike across network platforms. On the other hand, it is to be expected that the contention overhead will increase as the connectivity in the network decreases. This is also confirmed for all the kernels.

5.1 IS

For this kernel, there is a significant deviation from the ideal curve for all three platforms (see Figure 3). The overheads may be analyzed by considering the different modes of execution. In this kernel, NORMAL and MUTEX are the only significant modes of execution (see Figure 7). The network accesses in the NORMAL mode are for ordinary data transfer, and the accesses in MUTEX are for synchronization. The latency and contention overheads incurred in the MUTEX mode is higher than in the NORMAL mode (see Figures 8 and 9). As a result of this, the total execution time in the MUTEX mode surpasses that in the NORMAL mode beyond a certain number of processors (see Figure 7), which also explains the dip in the speedup curve for mesh (see Figure 3).

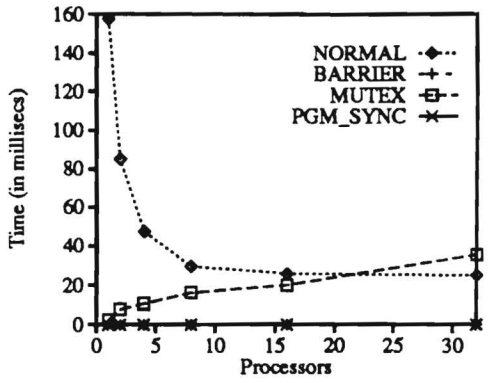

Figure 7: IS: Mode-wise Execn. Time (Mesh)

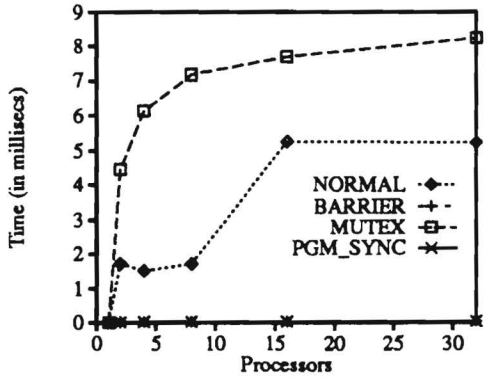

Figure 8: IS: Mode-wise Latency (Mesh)

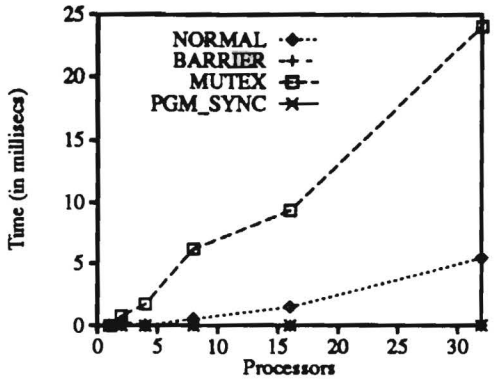

Figure 9: IS: Mode-wise Contention (Mesh)

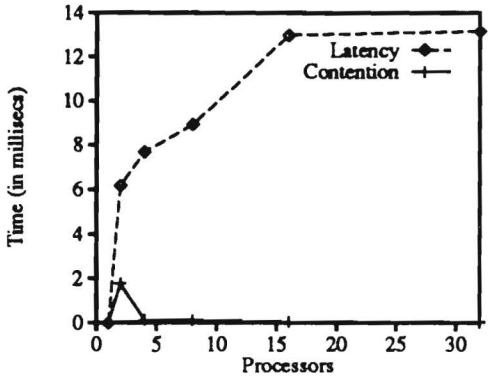

Figure 10: IS: Latency and Contention (Full)

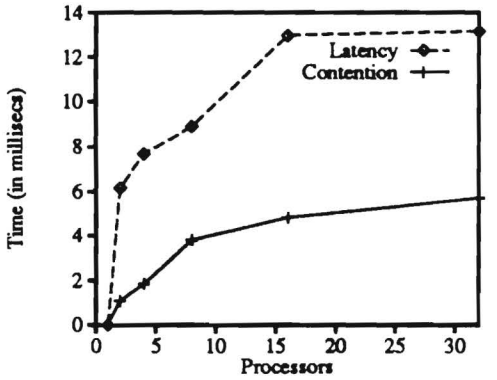

Figure 11: IS: Latency and Contention (Cube)

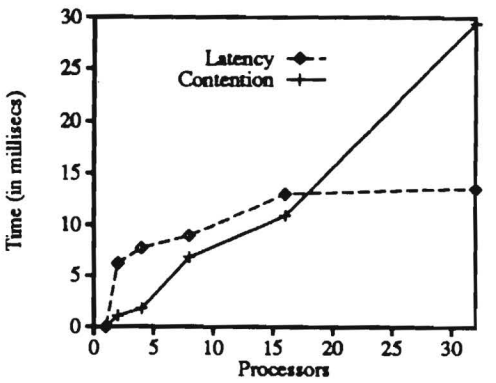

Figure 12: IS: Latency and Contention (Mesh)

Figures 10, 11 and 12. show the latency and contention overheads for the three hardware platforms. In IS, since every processor needs to access the data of all other processors, and since the data is equally partitioned among the executing processors, the number of accesses to remote locations grows as $(p-1)/p$. This explains the flattening of the latency overhead curve for all three network platforms as p increases. On the mesh network the contention overhead surpasses the latency overhead at around 18 processors. Table 1 summarizes the overheads for IS obtained by interpolating the datapoints from our simulation results.

IS	Full	Cube	Mesh
Comp. Time (ms)	$129.3/p^{0.7}$	$129.3/p^{0.7}$	$129.3/p^{0.7}$
Latency (ms)	$13.2(1-\frac{1}{p})$	$13.2(1-\frac{1}{p})$	$13.2(1-\frac{1}{p})$
Contention (ms)	*Negligible*	$4.0 \log p$	$0.9p$

Table 1: IS : Overhead Functions

Parallelization of this kernel increases the amount of work to be done for a given problem size (see [22]). This inherent algorithmic overhead causes a deviation of the ideal curve from the linear curve (see Figure 3). This is also confirmed in Table 1, where the computation time does not decrease linearly with the number of processors. This indicates the kernel is not scalable for small problem sizes. As can be seen from Table 1, the contention overhead is negligible and the latency overhead converges to a constant with a sufficiently large number of processors on a fully connected network. Thus for a fully connected network, the scalability of this kernel is expected to closely follow the ideal curve. For the cube and mesh platforms, the contention overhead grows logarithmically and linearly with the number of processors, respectively. Therefore, the scalability of IS on these two platforms is likely to be worse than for the fully connected network. From the above observations, we can conclude that IS is not very scalable for the chosen problem size on the three hardware platforms. However, if the problem is scaled up, the coefficient associated with the computation time will increase thus making IS more scalable.

5.2 FFT

The algorithmic and interaction overheads for the FFT kernel are marginal. Thus the real execution curves for all three platforms as well as the ideal curve are close to the linear one as shown in Figure 4. The execution time is dominated by the NORMAL mode (Figure 13). The latency and contention overheads (Figures 14 and 15) incurred in this mode are insignificant compared to the total

execution time, despite the growth of contention overhead with increasing number of processors.

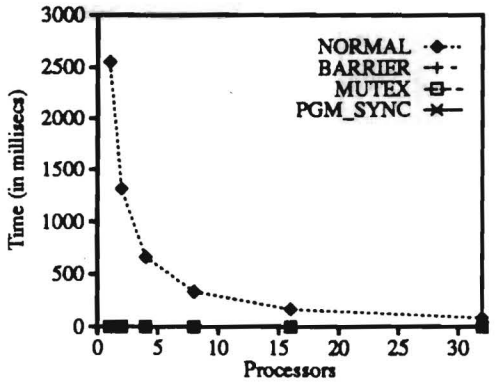

Figure 13: FFT: Mode-wise Execn. Time (Mesh)

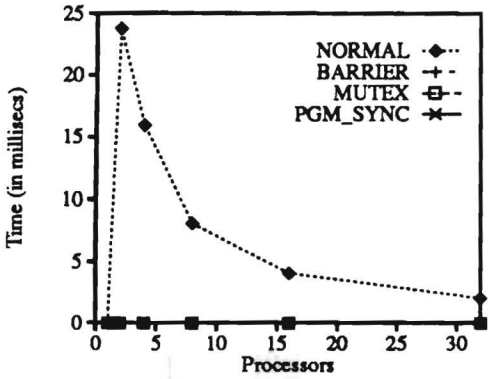

Figure 14: FFT: Mode-wise Latency (Mesh)

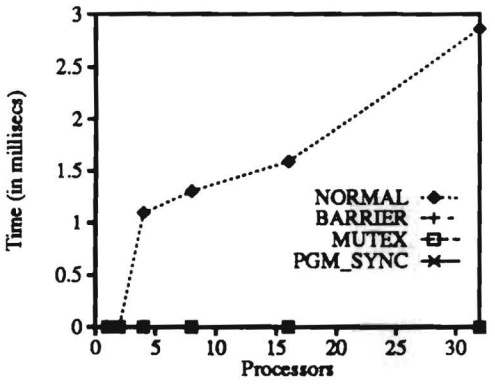

Figure 15: FFT: Mode-wise Contention (Mesh)

The communication in FFT has been optimized as suggested in [8] into a single phase where every processor accesses the data of all the other processors in a skewed manner. The number of such non-local accesses incurred by a processor grows as $O((p-1)/p^2)$ with the number of processors, and the latency overhead curves for all three networks reflect this behavior. As a result of skewing the communication among the processors, the contention is negligible on the full (Figure 16) and the cube (Figure 17) platforms. On the mesh (Figure 18), the contention surpasses the latency overhead at around 28 processors. Table 2 summarizes the overheads for FFT obtained by interpolating the datapoints from our simulation results.

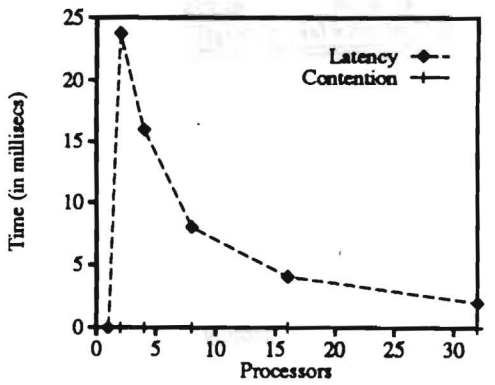

Figure 16: FFT: Latency and Contention (Full)

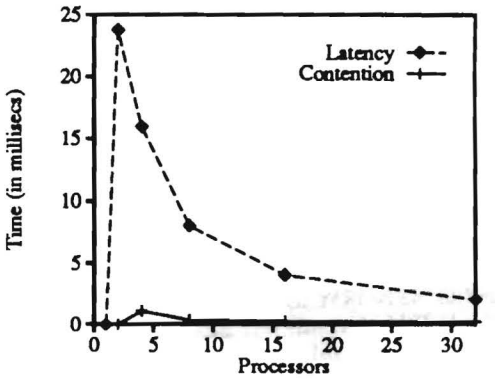

Figure 17: FFT: Latency and Contention (Cube)

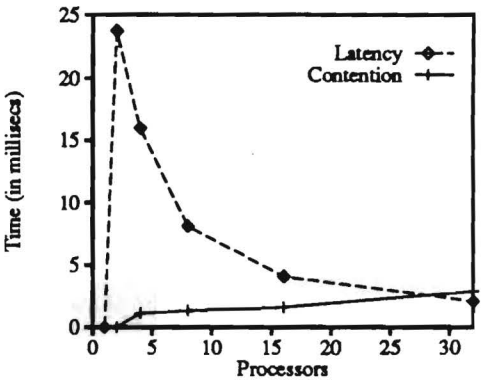

Figure 18: FFT: Latency and Contention (Mesh)

With marginal algorithmic overheads and decreasing number of messages exchanged per processor (latency overhead), the contention overhead is the only artifact that can cause deviation from linear behavior. But with skewed communication accesses, the contention overhead has also been minimized and begins to show only on the mesh network where it grows linearly (see Table 2). Thus we can conclude that the FFT kernel is scalable for the fully-connected and cube platforms. For the mesh platform, it would take 200 processors before the contention overhead starts dominating for the

FFT	Full	Cube	Mesh
Comp. Time (s)	$2.5/p$	$2.5/p$	$2.5/p$
Latency (ms)	$49.9/p^{0.9}$	$49.9/p^{0.9}$	$49.9/p^{0.9}$
Contention (us)	*Negligible*	*Small*	$63.5p$

Table 2: FFT : Overhead Functions

64K problem size. With increase in problem size (N), the local computation that performs a radix-2 Butterfly is expected to grow as $O((N/p)\log(N/p))$ while the communication for a processor is expected to grow as $O(N(p-1)/p^2)$. Hence, increase in data size will increase its scalability on all hardware platforms.

5.3 CHOLESKY

The algorithmic overheads for CHOLESKY cause a significant deviation from linear behavior for the ideal curve as shown in Figure 6. An examination of the execution times (Figure 19) shows that the bulk of the time is spent in the NORMAL mode which performs the actual factorization. The communication overheads in the NORMAL mode for the data accesses of the sparse matrix outweigh the accesses for synchronization variables (Figures 20 and 21). Thus the time spent in the MUTEX mode (which represents dynamic scheduling and accesses to critical sections) is insignificant compared to the NORMAL mode Although, the contention overhead in the NORMAL mode increases quite rapidly with the number of processors the overall impact of communication on the execution time is insignificant (see Figure 19).

As with FFT, the number of non-local memory accesses made by a processor decreases with increasing number of processors explaining a decreasing latency overhead. The contention overhead is negligible for the fully-connected network (Figure 22) and grows with increasing processors for the cube (Figure 23), becoming more dominant than the latency overhead for the mesh (Figure 24) at around 20 processors. Table 3 summarizes the overheads for CHOLESKY obtained by interpolating the datapoints from our simulation results.

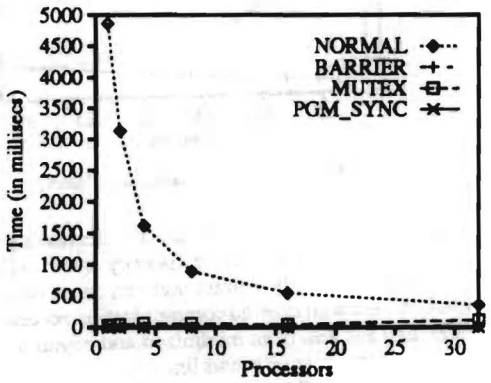

Figure 19: CHOLESKY: Mode-wise Execn. Time (Mesh)

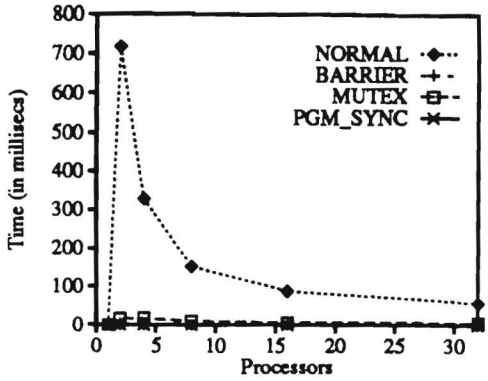

Figure 20: CHOLESKY: Mode-wise Latency (Mesh)

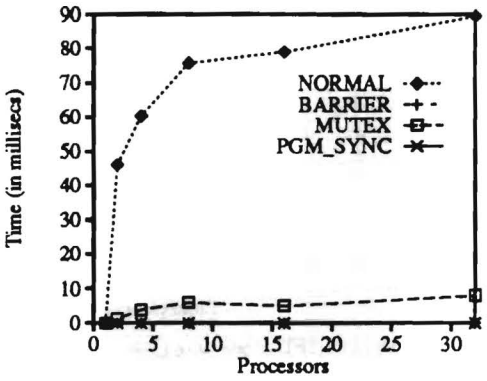

Figure 21: CHOLESKY: Mode-wise Contention (Mesh)

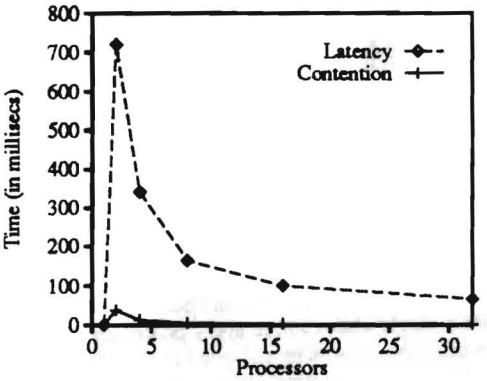

Figure 22: CHOLESKY: Latency and Contention (Full)

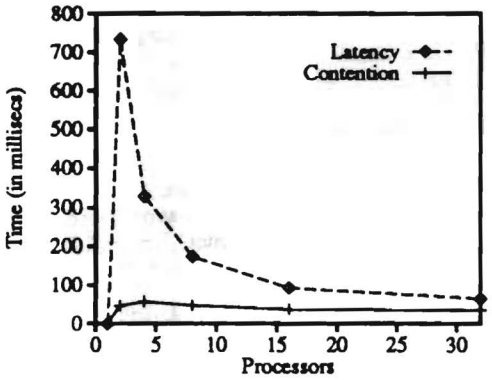

Figure 23: CHOLESKY: Latency and Contention (Cube)

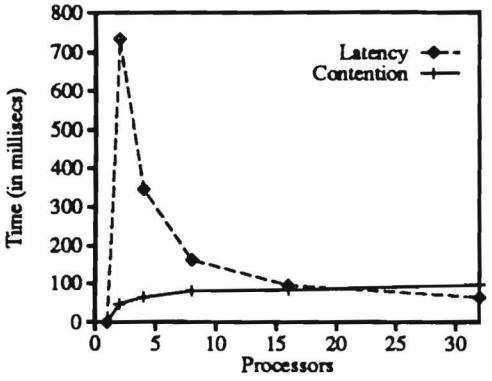

Figure 24: CHOLESKY: Latency and Contention (Mesh)

CHOLESKY	Full	Cube	Mesh
Comp. Time (s)	$3.9/p^{0.8}$	$3.9/p^{0.8}$	$3.9/p^{0.8}$
Latency (s)	$1.2/p^{0.9}$	$1.2/p^{0.9}$	$1.2/p^{0.9}$
Contention (ms)	*Negligible*	*Constant*	$39.9 \log p$

Table 3: CHOLESKY : Overhead Functions

The deviation of the ideal from the linear curve (Figure 6) indicates that the kernel is not very scalable for the chosen problem size due to the inherent algorithmic overhead as in IS. As can be observed from Table 3, the latency decreases with increasing number of processors and the scalability of the real execution would thus be dictated by the contention overhead. The contention on the fully-connected and cube networks is negligible thus projecting speedup curves that closely follow the ideal speedup curve for these platforms. On the other hand, the contention grows logarithmically on the mesh making this platform less scalable. With increasing problem sizes, the coefficient associated with the computation time in the above table is likely to grow faster than the coefficients associated with the communication overheads (verified by experimentation). Hence, an increase in problem size would enhance the scalability of this kernel on all hardware platforms.

6 Concluding Remarks

We used an execution-driven simulation platform to study the scalability characteristics of EP, IS, FFT, CG, and CHOLESKY on three shared memory platforms, respectively, with a fully-connected, cube, and mesh interconnection networks. The simulator allows for the separation of the algorithmic and interaction overheads in a parallel system. Separating the overheads provided us with some key insights into the algorithmic characteristics and architectural features that limit the scalability for these parallel systems. Algorithmic overheads such as the additional work incurred in parallelization could be a limiting factor for scalability as observed in IS and CHOLESKY. In shared memory machines with private caches, as long as the applications are well-structured to exploit locality, the key determinant to scalability is network contention. This is particularly true for most commercial shared memory multiprocessors which have sufficiently large caches.

We have illustrated the usefulness as well as the feasibility of our top-down approach for understanding the scalability of parallel systems. This approach can be used to study the impact of other system parameters (such as link bandwidth and processor speed) on scalability and provide guidelines for application design as well as evaluate architectural design decisions.

References

[1] A. Agarwal. Limits on Interconnection Network Performance. *IEEE Transactions on Parallel and Distributed Systems*, 2(4):398–412, October 1991.

[2] G. M. Amdahl. Validity of the Single Processor Approach to achieving Large Scale Computing Capabilities. In *Proceedings of the AFIPS Spring Joint Computer Conference*, pages 483–485, April 1967.

[3] T. E. Anderson. The Performance of Spin Lock Alternatives for Shared-Memory Multiprocessors. *IEEE Transactions on Parallel and Distributed Systems*, 1(1):6–16, January 1990.

[4] D. Bailey et al. The NAS Parallel Benchmarks. *International Journal of Supercomputer Applications*, 5(3):63–73, 1991.

[5] E. A. Brewer, C. N. Dellarocas, A. Colbrook, and W. E. Weihl. PROTEUS : A high-performance parallel-architecture simulator. Technical Report MIT-LCS-TR-516, Massachusetts Institute of Technology, Cambridge, MA 02139, September 1991.

[6] D. Chen, H. Su, and P. Yew. The Impact of Synchronization and Granularity on Parallel Systems. In *Proceedings of the 17th Annual International Symposium on Computer Architecture*, pages 239–248, 1990.

[7] R. G. Covington, S. Madala, V. Mehta, J. R. Jump, and J. B. Sinclair. The Rice parallel processing testbed. In *Proceedings of the ACM SIGMETRICS 1988 Conference on Measurement and Modeling of Computer Systems*, pages 4–11, Santa Fe, NM, May 1988.

[8] D. Culler et al. LogP : Towards a realistic model of parallel computation. In *Proceedings of the 4th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, pages 1–12, May 1993.

[9] R. Cypher, A. Ho, S. Konstantinidou, and P. Messina. Architectural requirements of parallel scientific applications with explicit communication. In *Proceedings of the 20th Annual*

International Symposium on Computer Architecture, pages 2–13, May 1993.

[10] S. J. Eggers and R. H. Katz. The Effect of Sharing on the Cache and Bus Performance of Parallel Programs. In *Proceedings of the Third International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 257–270, Boston, Massachusetts, April 1989.

[11] J. L. Gustafson, G. R. Montry, and R. E. Benner. Development of Parallel Methods for a 1024-node Hypercube. *SIAM Journal on Scientific and Statistical Computing*, 9(4):609–638, 1988.

[12] A. H. Karp and H. P. Flatt. Measuring Parallel processor Performance. *Communications of the ACM*, 33(5):539–543, May 1990.

[13] V. Kumar and V. N. Rao. Parallel Depth-First Search. *International Journal of Parallel Programming*, 16(6):501–519, 1987.

[14] F. H. McMahon. The Livermore Fortran Kernels : A Computer Test of the Numerical Performance Range. Technical Report UCRL-53745, Lawrence Livermore National Laboratory, Livermore, CA, December 1986.

[15] J. M. Mellor-Crummey and M. L. Scott. Algorithms for Scalable Synchronization on Shared-Memory Multiprocessors. *ACM Transactions on Computer Systems*, 9(1):21–65, February 1991.

[16] G. F. Pfister and V. A. Norton. Hot Spot Contention and Combining in Multistage Interconnection Networks. *IEEE Transactions on Computer Systems*, C-34(10):943–948, October 1985.

[17] S. K. Reinhardt et al. The Wisconsin Wind Tunnel : Virtual prototyping of parallel computers. In *Proceedings of the ACM SIGMETRICS 1993 Conference on Measurement and Modeling of Computer Systems*, pages 48–60, Santa Clara, CA, May 1993.

[18] E. Rothberg, J. P. Singh, and A. Gupta. Working sets, cache sizes and node granularity issues for large-scale multiprocessors. In *Proceedings of the 20th Annual International Symposium on Computer Architecture*, pages 14–25, May 1993.

[19] Jaswinder Pal Singh, Wolf-Dietrich Weber, and Anoop Gupta. SPLASH: Stanford Parallel Applications for Shared-Memory. Technical Report CSL-TR-91-469, Computer Systems Laboratory, Stanford University, 1991.

[20] A. Sivasubramaniam, U. Ramachandran, and H. Venkateswaran. Message-Passing: Computational Model, Programming Paradigm, and Experimental Studies. Technical Report GIT-CC-91/11, College of Computing, Georgia Institute of Technology, February 1991.

[21] A. Sivasubramaniam, G. Shah, J. Lee, U. Ramachandran, and H. Venkateswaran. Experimental Evaluation of Algorithmic Performance on Two Shared Memory Multiprocessors. In Norihisa Suzuki, editor, *Shared Memory Multiprocessing*, pages 81–107. MIT Press, 1992.

[22] A. Sivasubramaniam, A. Singla, U. Ramachandran, and H. Venkateswaran. An Approach to Scalability Study of Shared Memory Parallel Systems. Technical Report GIT-CC-93/62, College of Computing, Georgia Institute of Technology, October 1993.

[23] A. Sivasubramaniam, A. Singla, U. Ramachandran, and H. Venkateswaran. Machine Abstractions and Locality Issues in Studying Parallel Systems. Technical Report GIT-CC-93/63, College of Computing, Georgia Institute of Technology, October 1993.

[24] A. Sivasubramaniam, A. Singla, U. Ramachandran, and H. Venkateswaran. A Simulation-based Scalability Study of Parallel Systems. Technical Report GIT-CC-93/27, College of Computing, Georgia Institute of Technology, April 1993.

[25] X-H. Sun and J. L. Gustafson. Towards a better Parallel Performance Metric. *Parallel Computing*, 17:1093–1109, 1991.

[26] J. C. Wyllie. *The Complexity of Parallel Computations*. PhD thesis, Department of Computer Science, Cornell University, 1979.

JOURNAL OF PARALLEL AND DISTRIBUTED COMPUTING 22, 411–426 (1994)

A Simulation-Based Scalability Study of Parallel Systems[1]

ANAND SIVASUBRAMANIAM, AMAN SINGLA, UMAKISHORE RAMACHANDRAN, AND H. VENKATESWARAN

College of Computing, Georgia Institute of Technology, Atlanta, Georgia 30332-0280

Scalability studies of parallel architectures have used scalar metrics to evaluate their performance. Very often, it is difficult to glean the sources of inefficiency resulting from the mismatch between the algorithmic and architectural requirements using such scalar metrics. Low-level performance studies of the hardware are also inadequate for predicting the scalability of the machine on real applications. We propose a top-down approach to scalability study that alleviates some of these problems. We characterize applications in terms of the frequently occurring kernels and their interaction with the architecture in terms of overheads in the parallel system. An overhead function is associated with each artifact of the parallel system that limits its scalability. We present a simulation platform called SPASM (Simulator for Parallel Architectural Scalability Measurements) that quantifies these overhead functions. SPASM separates the algorithmic overhead into its components (such as serial and work-imbalance overheads), and interaction overhead into its components (such as latency and contention). Such a separation is novel and has not been addressed in any previous study using real applications. We illustrate the top-down approach by considering a case study in implementing three NAS parallel kernels on two simulated message-passing platforms.

1. INTRODUCTION

With the recent rapid advances in technology, the last decade has witnessed a proliferation of parallel machines, both in industry and in academia. Coupled with this evolution there has also been a growing effort in the computer science community to go beyond pure algorithmic work to actual experimentation with parallel algorithms on real machines. Algorithmic work has usually been based on abstract models of parallel machines that may not accurately capture the features of the architecture that are important from the performance standpoint. While machine models used in sequential algorithm design have been extremely successful in predicting the running time on uniprocessors within a constant factor, experimentation has revealed that parallel systems[2] do not enjoy the same luxury due to additional degress of freedom. Analytical models for parallel systems are even more difficult to build and often use simplistic assumptions about the system to keep the complexity of such models tractable. *Scalability* is a notion frequently used to signify the "goodness" of parallel systems. A good understanding of this notion may be used to select the best algorithm–architecture combination for a problem, identify algorithmic and architectural bottlenecks, predict the performance of an algorithm on an architecture with a larger number of processors, determine the optimal number of processors to be used for the algorithm and the maximum speedup that can be obtained, and glean insight on the influence of the algorithm on the architecture and vice-versa to enable us to understand the scalability of other algorithm–architecture pairs.

Performance metrics such as speedup [2], scaled speedup [11], sizeup [28], experimentally determined serial fraction [14], and isoefficiency function [15] have been proposed for quantifying the scalability of parallel systems. While these metrics are extremely useful for tracking performance trends, they do not provide information adequate for understanding the reason why an algorithm does not scale well on an architecture. An understanding of the interaction between the algorithmic and architectural characteristics of a parallel system can give us such information. Studies undertaken by Kung [16] and Jamieson [13] help identify some of these characteristics from a theoretical perspective, but they do not provide any means of quantifying their effects.

Several performance studies address issues such as latency, contention, and synchronization. The limits on interconnection network performance [1, 21] and the scalability of synchronization primitives supported by the hardware [3, 19] are examples of such studies undertaken over the years. While such issues are extremely important, it is necessary to put the impact of these factors into perspective by considering them in the context of overall application performance. There are studies that use real applications to address specific issues like the effect of sharing in parallel programs on the cache and bus performance [10] and the impact of synchronization and task granularity on parallel system performance [6]. Cypher *et al.* [9] identify the architectural requirements, such as floating point operations, communications, and input/output, for message-passing scientific applications. Rothberg *et al.* [23] conduct a similar study toward identifying the cache and memory size requirements for several applications. However, there have been very few

[1]This work has been funded in part by NSF Grants MIPS-9058430 and MIPS-9200005, and by an equipment grant from DEC.

[2]The term parallel system is used to denote an algorithm–architecture combination.

0743-7315/94 $6.00

attempts at quantifying the effects of algorithmic and architectural interactions in a parallel system.

Since real-life applications set standards for computing, it is meaningful to use the same applications for the evaluation of parallel systems. We call such an application-driven approach *top-down approach to scalability study*. The main thrust of this approach is to identify important algorithmic and architectural artifacts that impact the performance of a parallel system, understand the interaction between them, quantify the impact of these artifacts on the execution time of an application, and use these quantifications in studying the scalability of a parallel system (Section 2). To this end, we have developed a simulation platform called SPASM (Simulator for Parallel Architectural Scalability Measurements), which identifies different *overhead functions* that help quantify deviations from ideal behavior of a parallel system (Section 3).

The following are the important contributions of this work:

- We propose a top-down approach to the performance evaluation of parallel systems.
- We define the notion of overhead functions associated with the different algorithmic and architectural characteristics to quantify the scalability of parallel systems.
- We separate the algorithmic overhead into a *serial* component and a *work-imbalance* component. We also isolate the overheads due to *network latency* (the actual hardware transmission time of a message in the network) and *contention* (the amount of time spent in the network waiting for a resource to become free) from the overall execution time of an application. We are not aware of any other work that separates these overheads in the context of real applications and believe that such a separation is very important for understanding the interaction between algorithms and architectures.
- We design and implement a simulation platform that incorporates these methods for quantifying the overhead functions.

This work is part of a larger project which aims at understanding the significant issues in the design of scalable parallel systems using the above-mentioned top-down approach. In our earlier work, we studied issues such as task granularity, data distribution, scheduling, and synchronization by implementing frequently used parallel algorithms on shared memory [25] and message-passing [24] platforms. In [27], we illustrate the top-down approach for the scalability study of shared memory systems. In this paper, we conduct a similar study for message-passing systems. In a related paper [26] we evaluate the use of abstractions for the network and locality in the context of simulating cache-coherent shared memory multiprocessors.

We illustrate the top-down approach through a case study, implementing three NAS parallel kernels [4] on two message-passing platforms (a bus and a binary hypercube) simulated on SPASM. The algorithmic characteristics of these kernels are discussed in Section 4, details of the two architectural platforms are presented in Section 5, and the results of our study are summarized in Section 6. Based on these results we present the scalability implications for these two systems with respect to these kernels in Section 7. Concluding remarks and directions for future research are given in Sections 8 and 9.

2. TOP–DOWN APPROACH

Adhering to the RISC ideology in the evolution of sequential architectures, we use *real world applications* in the performance evaluation of parallel machines. However, applications normally tend to contain large volumes of code that are not easily portable, and a level of detail that is not very familiar to someone outside that application domain. Hence, computer scientists have traditionally used parallel algorithms that capture the interesting computation phases of applications for benchmarking their machines. Such abstractions of real applications that capture the main phases of the computation are called *kernels*. One can go even lower than kernels by abstracting the main *loops* in the computation (like the Lawrence Livermore loops [18]) and evaluating their performance. As one goes lower in the hierarchy, the outcome of the evaluation becomes less realistic. Even though an application may be abstracted by the kernels inside it, the sum of the times spent in the underlying kernels may not necessarily yield the time taken by the application. There is usually a cost involved in moving from one kernel to another such as the data movements and rearrangements in an application that are not part of the kernels that it is comprised of. For instance, an efficient implementation of a kernel may need to have the input data organized in a certain fashion which may not necessarily be the format of the output from the preceding kernel in the application. Despite its limitations, we believe that the scalability of an application with respect to an architecture can be captured by studying its kernels, since they represent the computationally intensive phases of an application. Therefore, we have used kernels in this study, in particular the NAS parallel kernels [4], that have been derived from a large number of computational fluid dynamics applications.

Parallel system overheads may be broadly classified into a purely algorithmic component (*algorithmic overhead*), and a component arising from the interaction of the algorithm and the architecture (*interaction overhead*). The algorithmic overhead is due to the inherent *serial* part [2] and the *work-imbalance* in the algorithm. Isolating these two components of the algorithmic overhead would help in re-structuring the algorithm to improve its performance. Algorithmic overhead is the difference between the linear curve and that which would be obtained by executing the algorithm on an ideal machine such as the PRAM [30] (the "ideal" curve in Fig. 1). Such a machine idealizes the parallel architecture by as-

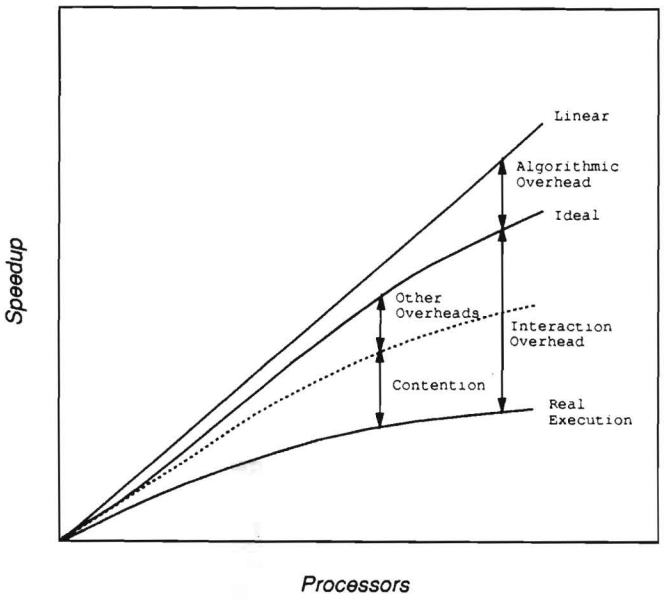

FIG. 1. Top-down approach to scalability study.

suming an infinite number of processors and unit costs for communication and synchronization. Hence, the real execution could deviate significantly from the ideal execution due to overheads such as latency, contention, synchronization, scheduling, and cache effects. These overheads are lumped together as the interaction overhead. To fully understand the scalability of a parallel system it is important to isolate the influence of each component of the interaction overhead on the overall performance. For instance, in an architecture devoid of network contention, the communication pattern of the application would dictate the latency overhead, and its performance may lie between the ideal curve and the real execution curve (see Fig. 1).

The key elements of our top-down approach for studying the scalability of parallel systems are

- experiment with real world applications,
- identify parallel kernels that occur in these applications,
- study the interaction of these kernels with architectural features to separate and quantify the overheads in the parallel system, and
- use these overheads for predicting the scalability of parallel systems.

2.1. Implementing the Top–Down Approach

Scalability studies of parallel systems are complex due to the several degrees of freedom that exist in the systems. Experimentation, simulation, and analytical models are three techniques that have been commonly used for such studies. But each has its own limitations. We adopted the first technique in our earlier work by experimenting with frequently used parallel algorithms on shared memory [25] and message-passing [24] platforms. Experimentation is important and useful for scalability studies of existing architectures but has certain limitations. The underlying hardware is fixed, making it impossible to study the effect of changing individual architectural parameters, and it is difficult if not impossible to separate the effects of different architectural artifacts on the performance since we are constrained by the performance monitoring support provided by the parallel system. Further, monitoring program behavior via instrumentation can become intrusive, yielding inaccurate results.

Analytical models have often been used to give gross estimates for the performance of large parallel systems. In general, such models tend to make simplistic assumptions about program behavior and architectural characteristics to make the analysis using the model tractable. These assumptions restrict their applicability for capturing complex interactions between algorithms and architectures. For instance, models developed in [17, 29, 8] are mainly applicable to algorithms with regular communication structures that can be predetermined before execution of the algorithm. Madala and Sinclair [17] confine their studies to synchronous algorithms while [29] and [8] develop models for regular iterative algorithms. However, there exist several applications [23] with irregular data access, communication, and synchronization characteristics which cannot always be captured by such simple parameters. Further, an application may be structured to hide a particular overhead such as latency by overlapping computation with communication. It may be difficult to capture such dynamic program behavior using analytical models. Similarly, several other models make assumptions about architectural characteristics. For instance, the model developed in [20] ignores data inconsistency that can arise in a cache-based multiprocessor during the execution of an application and thus does not consider the coherence traffic on the network.

The main focus of our top-down approach is quantifying the overheads that arise from the interaction between the kernels and architecture and their impact on the overall execution of the application. It is not clear that these overheads can be easily modeled by a few parameters. Therefore, we use simulation for quantifying and separating the overheads. Experimentation is used in conjunction with simulation to provide an understanding of the performance of real applications on real architectures, and to identify the interesting kernels that occur in these applications for subsequent use in simulation studies. Simulation also has its limitations. It may not always be possible to predict system scalability with simulation owing to resource (time and space) constraints. But we believe that simulation can complement the analytical technique by using the datapoints obtained from simulation, which are closer to reality, as a feedback to refine existing models for predicting the scalability of larger systems. Further, our simulator can also be used to validate existing analytical models using real applications.

Our simulation platform (SPASM) provides an elegant set of mechanisms for quantifying the different overheads discussed earlier. The algorithmic overhead is quantified by computing the time taken for execution of a given parallel program on an ideal machine such as the PRAM [30] and measuring its deviation from a linear speedup curve. Further, we separate this overhead into that due to the serial part (*serial overhead*) and that due to work imbalance (*work-imbalance overhead*). The interaction overhead is also separated into its component parts. We currently do not address scheduling overheads by assuming that the number of processes spawned in a parallel program is equal to the number of processors in the simulated machine, and that a process is bound to a processor and does not migrate.[3] We have also confined ourselves to message-passing platforms in this study, where synchronization and communication are intertwined. Thus the interaction overhead is quantified using the *latency overhead function* and the *contention overhead function* that are described in the next section. In a shared memory platform, it would be interesting to consider the impact of communication and synchronization in the algorithm on latency and contention separately, but such issues are beyond the scope of this paper. For the rest of this paper, we confine ourselves to the only two aspects of the interaction overhead that are germane to this study, namely, latency and contention.

3. SPASM

SPASM is an execution-driven simulator that enables us to conduct a variety of scalability measurements of parallel applications on a number of simulated hardware platforms. SPASM has been written using CSIM, a process-oriented sequential simulation package, and currently runs on SPARCstations. SPASM provides support for process control, communication, and synchronization. The input to the simulator are parallel applications written in C, which are compiled and linked with the simulator libraries.

Architectural simulators have normally tended to be very slow, making it tedious to get a wide range of data points on realistic problems. Simulating the entire instruction set of a processor can slow down the simulation considerably. Since the thrust of this study is to understand interesting characteristics of parallel machines and their impact on the algorithm, instruction-level simulation is not likely to contribute extensively to this understanding. Hence, we have confined ourselves to simulating the interesting aspects of parallel machines. The bulk of the processor instruction streams is executed at the speed of the native processor (the SPARC in this case) and only those instructions that could potentially involve the network are simulated by SPASM. Examples of such instructions include sends and receives on a message-passing platform and loads and stores on a shared memory platform. Such instructions are trapped by our simulator and simulated exactly according to the semantics of these instructions on each particular platform. SPASM reconciles the real time with the simulated time using these trapped instructions. Upon such a trap, SPASM computes the time for the block of instructions that were executed at the native speed since the previous trap and updates the simulation clock of the processor. This strategy has considerably lowered the simulation time which is at most a factor of two compared to the real time for the applications considered. This strategy has also been used in other recent simulation studies [5, 7, 22].

The novel feature of our simulation study is the separation of overheads in a parallel system. Providing the functionality that is needed for such a separation requires a considerable amount of instrumentation. There exist simulators in the public domain (such as Proteus [5]) that provide a certain basic functionality. We have used CSIM as the starting point since the programming effort necessary to provide the needed functionality is comparable whether we use a basic process-based simulation package such as CSIM or any other architecture simulator.

Figure 2 depicts the architecture of our simulation platform. Each node in the parallel machine is abstracted by a processor (PE), a cache module (Cache) and a network interface (NI). These three entities are implemented as CSIM processes. The CSIM process representing a processor executes the code associated with the corresponding thread of control in the parallel program. These processes execute at the speed of the native processor until they encounter an instruction that needs to be trapped by the simulator like a load/store on a shared memory platform or a send/receive on a message-passing platform. On a shared memory platform, the processor issues load/store requests to its cache module. The cache module services the request, invoking the network interface if required. Since the shared memory platform is beyond the purview of this study, we do not discuss any further details of its implementation. On a message-passing platform, the processor interacts directly with the network interface. On a send, the processor creates a message (a data structure) and places it in a mailbox. The network interface picks up the message from the mailbox, determines the routing information, waits for the relevant links of the network to become free, accounts for the software and hardware overheads, and delivers the message to the mailbox of the destination processor. On a synchronous receive, the processor blocks until a message appears in its mailbox.

The system parameters that can be specified to SPASM are the *number of processors* (p), the *clock*

[3] We do not distinguish between the terms *process*, *processor*, and *thread*, and use them synonymously.

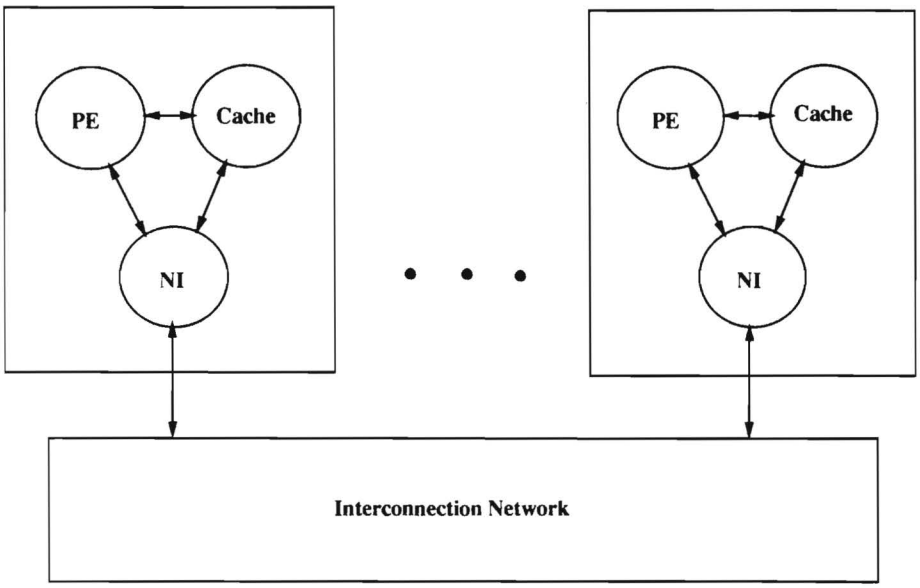

FIG. 2. Architecture of SPASM.

speed of the processor[4], the *hardware bandwidth* of the links in the network, the *switching delay*, the *software latency* for transmission of a message, and the sustained *software bandwidth*.

3.1. Metrics

SPASM provides a wide range of statistical information about the program execution. It gives the *total time* (simulated time) which is the maximum of the running times of the individual parallel processors. This is the time that would be taken by an execution of the parallel program on the target parallel machine. *Speedup* using p processors is measured as the ratio of the total time on 1 processor to the total time on p processors.

Ideal time is the total time taken by a parallel program to execute on an ideal machine such as the PRAM. It includes the algorithmic overhead but does not include the interaction overhead. SPASM simulates an ideal machine to provide this metric. As we mentioned in Section 2, the difference between the linear time and the ideal time gives the algorithmic overhead.

A processor may wait for an event (such as a synchronization or a communication operation) even before the event occurs. For the message passing platform being considered the only events are the sending and receiving of messages. If a receive is posted prior to the matching send, then the difference between the two times is due to skews between the processors and is called the *wait time* of a processor. In an ideal machine, the wait time is entirely due to the work-imbalance overhead and is a metric provided by SPASM. The difference between the algorithmic overhead and the work-imbalance overhead gives the serial overhead in the algorithm.

As mentioned in Section 2, we want to isolate the effects of latency and contention in the system. In a network with no contention, the overhead of a message would be purely due to software and hardware latencies for communication. Each processor performing a blocking receive is expected to see this latency when all other conditions are ideal. The sum of all these overheads incurred by a processor is called the *network latency* ($f_l(p)$). But this may not necessarily reflect the *real* latencies observed by a processor since some of it may be hidden by the overlap of computation and communication. We call the latency *observed* by a processor the *latency overhead function* ($f_L(p)$). SPASM gives the network latency of a processor as well as the latency overhead function seen by a processor. SPASM measures the latter entity by time-stamping messages at the sending processor. SPASM checks to see if the destination processor posted a receive for the message after it was sent, in which case only a corresponding part of the network latency is accounted for as the latency overhead. If the destination processor posted a wait for the message before it was sent, then the entire network latency is charged to it as the latency overhead.

As with latency, SPASM provides information about the *network contention* ($f_c(p)$) that a processor is supposed to incur and the *contention overhead function* ($f_C(p)$) actually observed by the processor at the receiving end. Network contention incurred by a processor is the sum of all the waiting times (due to network links not

[4]Even though we assume that each processor is a SPARC chip, we can vary the clock speed of the simulated chip which provides us with a convenient mechanism for varying the computation-to-communication ratio and studying the scalability of future systems built with faster processors.

being available) for all the messages that it receives. A processor may choose to hide a part of this contention by overlapping computation with communication, or a processor may simply find a message already available when it posts a receive (in which case it does not see any contention). The contention actually *observed* by a processor is called the *contention overhead function* ($f_C(p)$). SPASM calculates this overhead using time-stamped messages and the time that would have been taken by a message on a contention-free network (i.e., the network latency).

The wait time experienced by a processor on a real machine includes the work-imbalance overhead (a purely algorithmic characteristic), as well as processor skews introduced due to the latency and contention experienced by the earlier messages. Let us denote the wait times due to work-imbalance, latency, and contention by W_w, W_l, and W_c, respectively, and the wait times measured by SPASM on an ideal machine, on a real machine with a contention-free network, and on the real machine by W_i, W_f, and W_r, respectively, Then the component wait times can be computed using the following expressions:

$$W_w = W_i$$
$$W_l = W_f - W_i$$
$$W_c = W_r - W_f.$$

From the above discussion, it follows that

$$\text{Total Time} = \text{Ideal Time} + f_L(p) + f_C(p) + W_r.$$

SPASM also provides statistical information about the network. It gives the utilization of each link in the network and the average queue lengths of messages at any particular link. This information can be useful in identifying network bottlenecks and for comparing the merits of different networks. Thus the metrics identified by SPASM quantify the algorithmic overhead and the interesting components of the interaction overhead.

4. ALGORITHMIC CHARACTERISTICS

Kernels are abstractions of the major phases of computation in an application that account for the bulk of the execution time. A *parallel kernel* is characterized by the data access pattern, the synchronization pattern, the communication pattern, the computation granularity (which is the amount of work done between synchronization points), and the data granularity (which is the amount of data manipulated between synchronization points). The last two together define the task granularity of the parallel kernel. These attributes are as seen from the point of view of the individual processors implementing the parallel kernel. If the parallel kernel is implemented using the message-passing style, then the data access pattern becomes unimportant (except for any cache effects) since all data accesses are to private memory. Further, the synchronization is usually merged with the communication in such an implementation. On the other hand, if a shared memory programming style is used, the communication pattern is not explicit and gets merged with the data access pattern.

The Numerical Aerodynamic Simulation (NAS) program at NASA Ames has identified a set of kernels [4] that are representative of a number of large scale computational fluid dynamics codes. In this study, we consider three of these kernels for the purposes of illustrating the top-down approach using SPASM. In this section, we identify their characteristics in a message-passing style implementation.

EP is the "Embarrassingly Parallel" kernel that generates pairs of Gaussian random deviates and tabulates the number of pairs in successive square annuli. This problem is typical of many Monte Carlo simulation applications. The kernel is computation bound and has little communication among the processors. A large number of floating point random numbers is calculated and a sequence of floating point operations is performed on them. The computation granularity of this section of the code is considerably large and is linear in the number of random numbers (the problem size) calculated. A data size of 64K pairs of random numbers has been chosen in this study. The operation performed on a computed random number is totally independent of the other random numbers. The processor assigned to a random number can thus execute all the operations for that number without any external data. Hence the data granularity is meaningless for this phase of the computation. Toward the end of this computation phase, a few global sums are calculated by using a logarithmic reduce operation. In step i of the reduction, a processor receives data from another which is a distance 2^i away and performs an addition of the received value with a local value. The size of the data exchanged (data granularity) in these logarithmic communication steps is 4 bytes (an integer). The computation granularity between these communication steps can lead to work imbalance since the number of participating processors halves after each step of the logarithmic reduction. However, since the computation is a simple addition, it does not cause any significant imbalance for this kernel. The amount of local computation in the initial computation phase overshadows the communication performed by a processor, suggesting a near linear speedup curve on most machines (unless the processing speed is to reach unrealistic limits). Table I summarizes the characteristics of the EP kernel.

TABLE I
Characteristics of EP

Phase	Description	Comp. Gran.	Data Gran.
1	Local floating pt. opns.	Large	N/A
2	Global sum	Integer addition	4 bytes

IS is the "Integer Sort" kernel that uses bucket sort to rank a list of integers, which is an important operation in "particle method" codes. A list of 64K integers with 2K buckets is chosen for this study. The input data is equally partitioned among the processors. Each processor maintains its own copy of the buckets for the chunk of the input list that is allocated to it. Hence, an update to the buckets, for the chunk of data allocated to a processor, is an entirely local operation to the processor. This computation phase is again linear in the problem size but the granularity of the computation is not as intensive as in EP. The processing of each list element needs only the update (an integer addition) of the corresponding local bucket. The buckets are then merged using a logarithmic reduce operation and propagated back to the individual processors. The logarithmic operation takes place as in EP, the difference being in the computation granularity and the data granularity (size of the messages exchanged). The message size (data granularity) in the communication steps is 8 Kbytes (2K integers). The computation granularity of the reduction is not a simple addition as in EP, but involves an integer addition for each of the buckets. This can lead to a non-trivial algorithmic work imbalance, depending on the chosen bucket size. The data size is chosen to be 64 Kbytes with 2K buckets to illustrate this work imbalance. Each processor then uses the merged buckets to calculate the rank of an element in its chunk of the input list. This phase of the kernel exhibits the same characteristics as the first computation phase (updating the local buckets). Table II summarizes the characteristics of the IS kernel.

The "Conjugate Gradient" kernel, CG, uses the conjugate gradient method to estimate the smallest eigenvalue of a symmetric positive-definite sparse matrix with a random pattern of non-zeroes that is typical of unstructured grid computations. A sparse matrix of size 1400×1400 containing 100,300 nonzeros has been used in the study. This kernel lies between EP and IS with respect to computation to communication ratio requirements. The sparse matrix and the vectors are partitioned by rows assigning an equal number of contiguous rows to each processor. The kernel performs 25 iterations in trying to approximate the solution of a system of linear equations using the conjugate gradient method. Each iteration involves the calculation of a sparse matrix-vector product and two vector–vector dot products. These are the only operations that involve communication. The computation granularity between these operations is linear in the number of rows (the problem size) and involves a floating point addition and multiplication for each row. The vector–vector dot product is calculated by first obtaining the intermediate dot products for the elements in the vectors local to a processor. This is again a local operation with a computation granularity linear in the number of rows assigned to a processor with a floating point multiplication and addition performed for each row. A global sum of the intermediate dot products is calculated by a logarithmic reduce operation (as in EP) yielding the final dot product. The computation granularity in the reduction is a floating point addition and the data granularity is 8 bytes (size of a double precision number). For the computation of the matrix-vector product, each processor performs the necessary calculations for the rows assigned to it in the resulting matrix (which are also the same rows in the sparse matrix that are local to the processor). But the calculation involves the elements of the input vector that are not local to a processor. Hence before the computation, the different portions of the input vector present on different processors are merged globally using a logarithmic reduce operation and the complete vector is replicated on each processor. The matrix-vector operation can then be carried out with entirely local operations. The logarithmic reduce operation for the merging does not have any computational granularity, but the data granularity doubles after each step of the operation. Initially the size of the messages is equal to the number of rows present on each processor ($11200/p$ bytes for $1400/p$ double precision numbers where p is the number of processors). After each step, the size of this message doubles since a processor needs to send the data that it receives along with its own local data to a processor that is at a distance a power of 2 away. Table III summarizes the characteristics for each iteration of the CG kernel.

TABLE II
Characteristics of IS

Phase	Description	Comp. gran.	Data gran.
1	Local bucket updates	Small	N/A
2	Global bucket merge	Small	8K bytes
3	Local bucket updates	Small	N/A

5. ARCHITECTURAL CHARACTERISTICS

A uniprocessor architecture is characterized by processing power as indicated by clock speed, instruction sets, clocks per instruction, floating point capabilities, pipelining, on-chip caches, memory size and bandwidth, and input–output capabilities. Parallel architectures have many more degrees of freedom, making it difficult to study each artifact. Since uniprocessor architecture is becoming standardized with the advent of RISC technology, we fix most of the processor characteristics by using the SPARC chip as the baseline for each processor in a parallel system. Such an assumption enables us to make a fair comparison of the relative merits of the interesting parallel architectural characteristics across different platforms. Input–output characteristics are beyond the purview of this study.

To illustrate the top-down approach, we use two message-passing architectures with different interconnection

TABLE III
Characteristics of CG

Phase	Description	Comp. Gran.	Data Gran.
1	Local floating pt. opns	Medium	N/A
2	Matrix–vector product		
	(a) Global vector merge	N/A	$(11200/p) * 2^i$ in step i
	(b) Local matrix–vector product	Medium	N/A
3	Vector–vector dot product		
	(a) Local vector–vector dot product	Small	N/A
	(b) Global sum	Floating pt. addition	8 bytes
4	Local floating pt. opns	Medium	N/A
5	Same as phase 3		
6	Local floating pt. opns	Medium	N/A

topologies: the *bus* and the *binary hypercube*. The bus platform consists of a number of nodes that are connected by a single 64-bit wide bus. Each processor in a node consists of a SPARC processor with local memory. The bus is an asynchronous Sequent-like bus (split transaction) with a cycle time of 150 nanoseconds. The cube platform closely resembles an iPSC/860 in terms of its communication capabilities and uses the *e*-cube routing algorithm. The nodes are connected by serial links with a bandwidth of 2.8 Mbytes/sec in a binary hypercube topology. Message transmission uses a circuit-switched wormhole routing strategy. We have chosen these two platforms because they provide very different communication characteristics. The bus provides a much higher bandwidth compared to a single link of the cube, but the latter is expected to provide more contention-free transmission due to its multiple links. The software overhead incurred is 100 microseconds per message, which is in keeping with existing message-passing machines.

Both platforms provide an identical message-passing interface to the programmer. They support blocking and non-blocking modes of message transfer. The semantics of these modes are the same as those available on an iPSC/860 [12]. A blocking send blocks the sender until the message has left the sending buffer. Such a send does not necessarily imply that the message has reached the destination processor or even entered the network. A blocking receive blocks until the message from a corresponding send is completely in the receiving buffer. A non-blocking send does not guarantee that the message has even left the user buffer and a non-blocking receive returns immediately to the user program even if the message has not been received.

Many message-passing parallel programs are easier to write if the underlying system provides *typed*-messages and selective blocking on *typed*-messages. Typed-messages make it easier to order messages instead of leaving the burden to the programmer. Both our platforms support this elegant facility. On a message receive, the processor picks up messages from its mailbox and queues them up until it finds a message of the type that it needs.

6. PERFORMANCE RESULTS

The simulations have been carried out for the execution of the three NAS kernels on the two message-passing platforms. We report results for two different processor speeds, one at the native SPARC speed and the second at 10 times the native SPARC speed. Since the main focus of this paper is an approach to scalability study, we have not dwelled on the scalability of parallel systems with respect to specific architectural artifacts to any great extent in this paper.

Figures 3, 4, and 5 show the speedups of the three NAS kernels on the two hardware platforms. The curves labeled "ideal" in these figures have been calculated using the ideal time given by SPASM. The curves show the maximum possible speedup that can be obtained for the given parallel program (a purely algorithmic characteristic). As explained by the characteristics of these kernels in Section 4, the "ideal" curve is observed to be almost linear for the EP kernel (Fig. 3) and close to linear for the CG kernel (Fig. 5) up to 64 processors. For the IS kernel

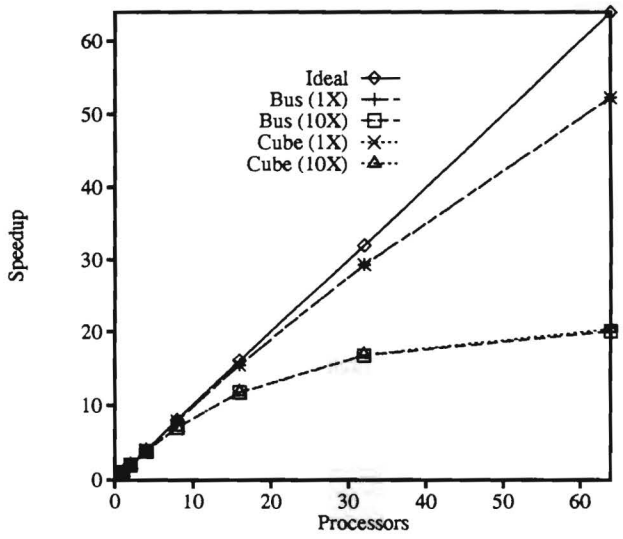

FIG. 3. EP: Speedup.

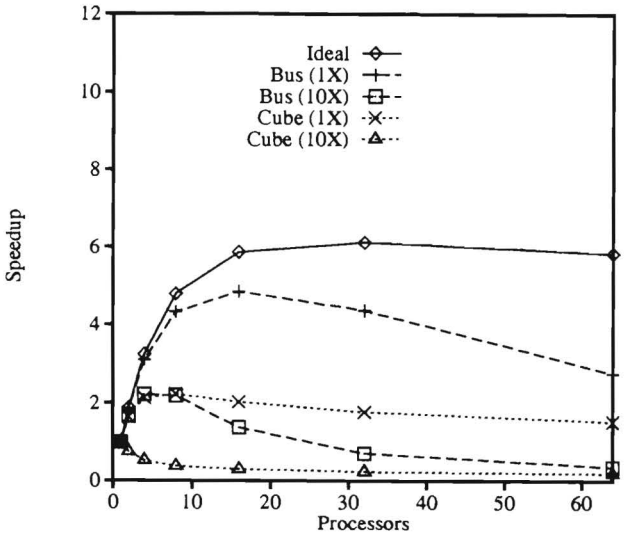

FIG. 4. IS: Speedup.

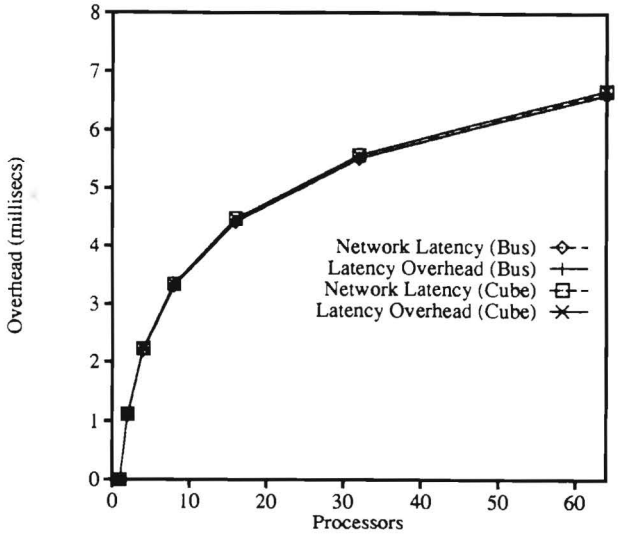

FIG. 6. EP: Latency.

(Fig. 4) with the given problem size, the work imbalance in the program dominates, yielding maximum performance at around 30 processors. A further increase in the number of processors results in a slowdown. The architectural overheads arise due to communication in the problem and result in a deviation from the algorithmically predicted speedup curve (labeled "ideal"). EP has a high computation-to-communication ratio, thus yielding a speedup curve (for both bus and cube) close to the ideal speedup with the processor running at SPARC speed (1X). CG is more communication bound, showing speedup curves that are significantly worse than the ideal speedup curve. The deviations from the ideal curve for IS lie between that for EP and CG. For this problem, the speedup curves are limited more by the algorithm than by the architectural overheads. Increasing the processing speed to 10 times the SPARC speed (10X) progressively reduces the computation-to-communication ratio for all kernels, thus yielding worse speedup curves (see corresponding 1X and 10X curves in Figs. 3, 4, and 5). The EP kernel, which uses short messages (4 bytes) for its prefix computations, shows practically no difference in speedups between the bus and cube platforms. On the other hand, the poor point-to-point bandwidth of the cube compared to the bus plays an important role in degrading the performance of the other two kernels which send messages of larger lengths (see Bus 1X and Cube 1X curves in Figs. 3 and 4).

Figures 6, 7, and 8 show the latency overhead of the architecture on the respective kernels with the processor running at the native SPARC speed. These curves have been drawn for the processor that observes the maximum

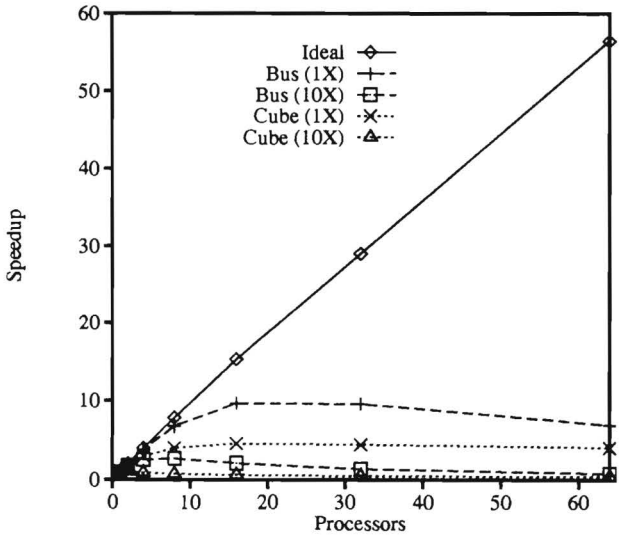

FIG. 5. CG: Speedup.

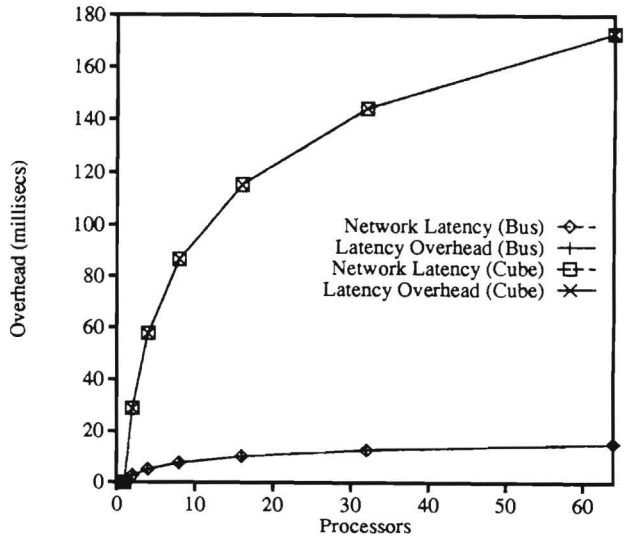

FIG. 7. IS: Latency.

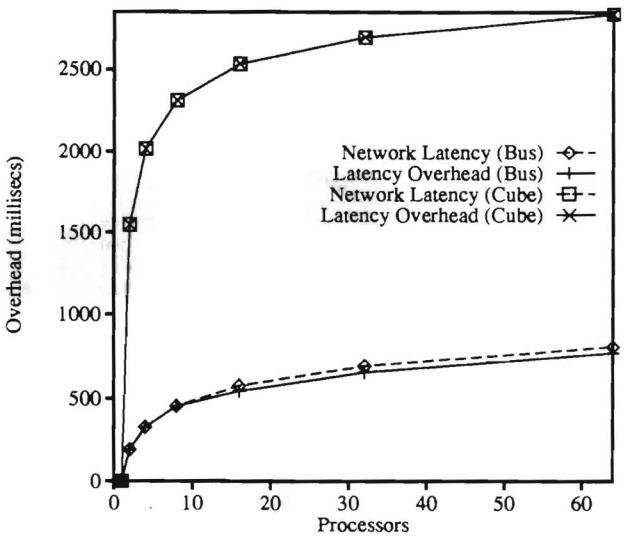

FIG. 8. CG: Latency.

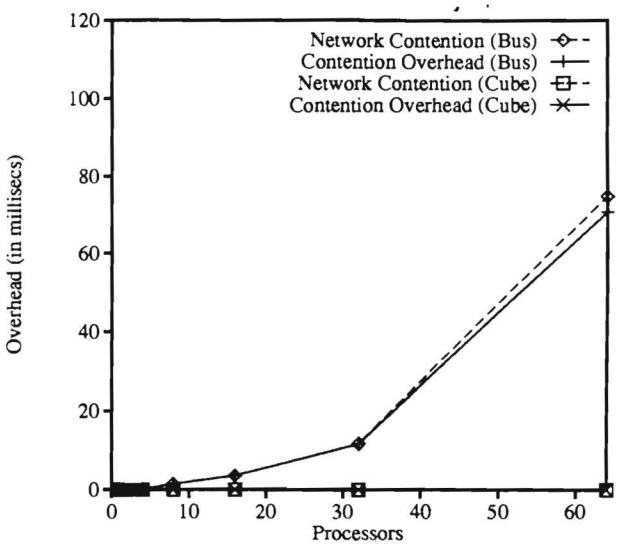

FIG. 10. IS: Contention.

latency in each case. In all the kernels, the network latency ($f_l(p)$) of a processor is almost identical to the latency overhead function ($f_L(p)$) observed by a processor, indicating that there is minimal overlap of computation with communication. The communication in all three kernels occurs only in the logarithmic reduce operations. The difference between them is in the size of messages exchanged in this operation and the bandwidth of the interconnect. Since the number of messages received by a processor grows logarithmically with the number of processors, all curves show a logarithmic behavior. The curves for latencies on the bus and the cube (see Fig. 6) are almost identical for EP. This is due to the short messages (4 bytes) used by EP for its data exchanges. The software overhead of 100 microseconds per message on both platforms is the more dominating factor, obviating the difference in the two hardware bandwidths. On the other hand, for IS and CG, which send longer messages, there is a considerable disparity between the bus and cube for network latency (see Figs. 7 and 8).

Figures 9, 10, and 11 show the contention overhead of the architectures for the respective kernels with the processor running at the native SPARC speed. A logarithmic reduce operation exchanges messages between processors that are separated by a distance that is a power of two. Such an operation can be elegantly mapped on the cube to be entirely contention free. On the other hand, all the messages have to be sequentially handled on the bus, giving rise to growing contention with increasing number of processors. As with latency, the network contention curves ($f_c(p)$) and the contention overhead curves ($f_C(p)$) are almost identical for the three kernels. There is negligible hiding of contention due to overlap of computation with communication. Only IS exhibits any hidden con-

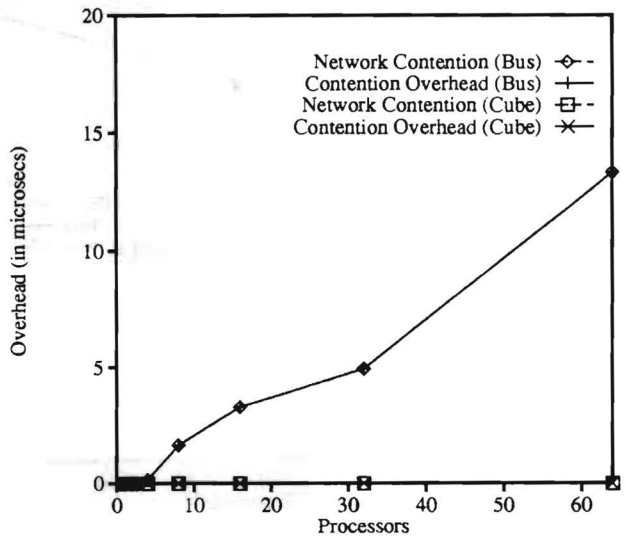

FIG. 9. EP: Contention.

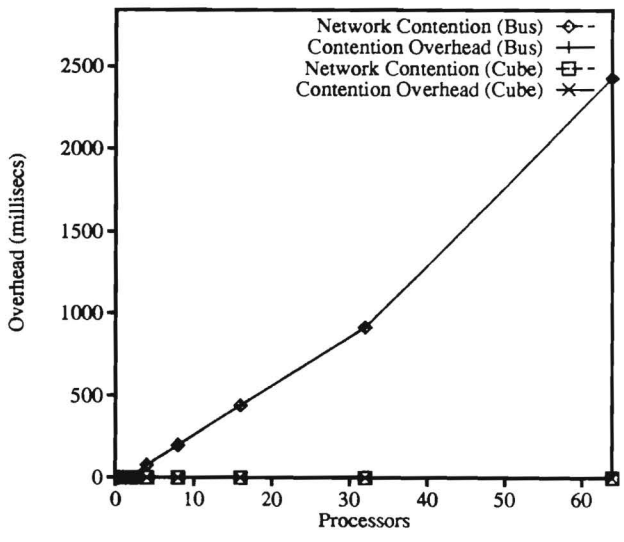

FIG. 11. CG: Contention.

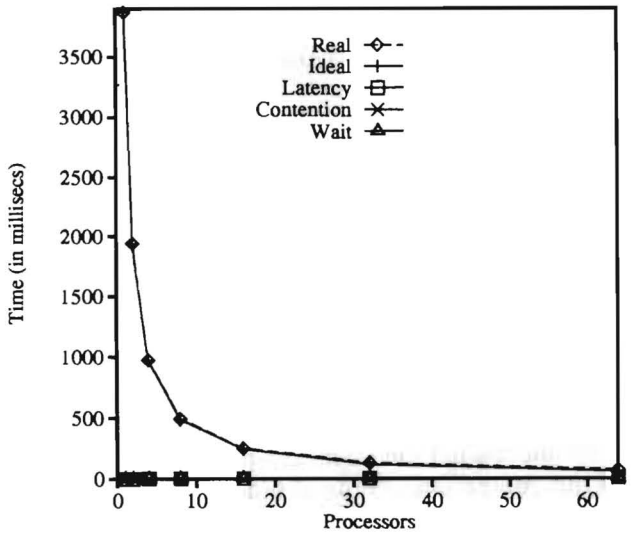

FIG. 12. EP: Overheads on bus.

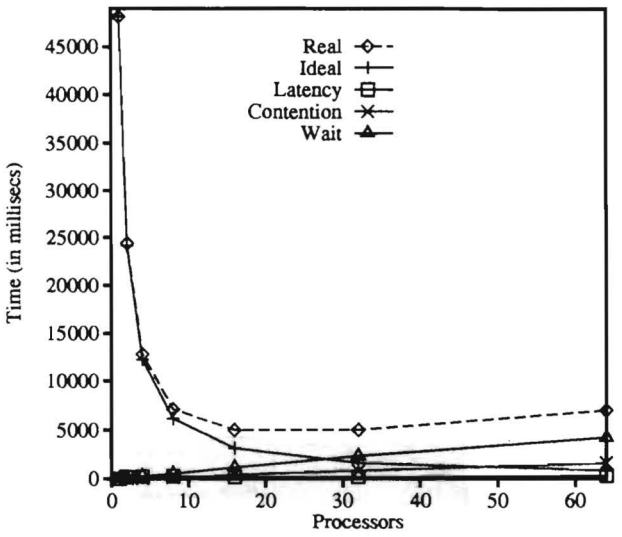

FIG. 14. CG: Overheads on bus.

tention for the 64-processor case (around 5% of the overall contention). The shape of the curves shows that the contention overhead on the bus grows faster than linear for all three kernels. Latency, which is a logarithmically growing function, is soon overtaken by the faster than linear growing contention function (at around 40 processors for IS and at around 12 processors for CG).

Figures 12, 13, and 14 show the breakup of times due to algorithmic and interaction overheads for the three kernels on the bus. Figures 15, 16, and 17 depict the same information for the cube. The timings shown are for a representative processor which executes the workload that is characteristic of the specific kernel. Note that this may not necessarily be the one that takes the longest time nor the one that experiences the maximum overheads. It is the processor that spends the maximum time for com-

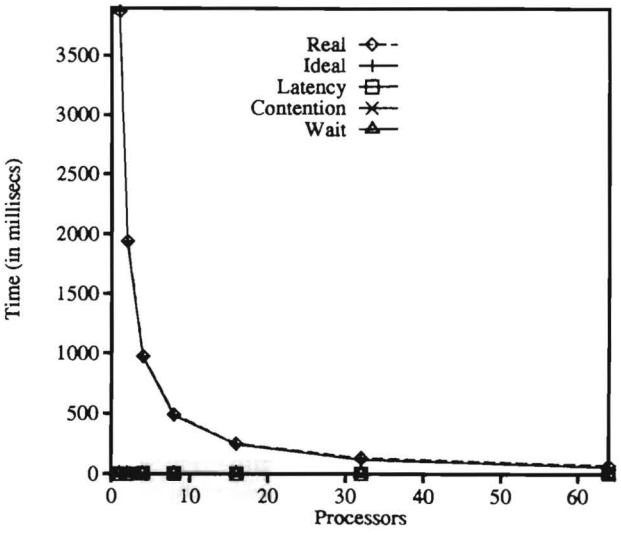

FIG. 15. EP: Overheads on cube.

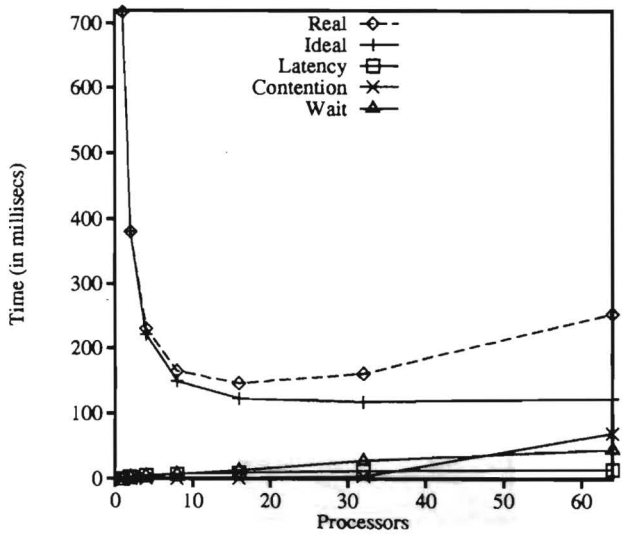

FIG. 13. IS: Overheads on bus.

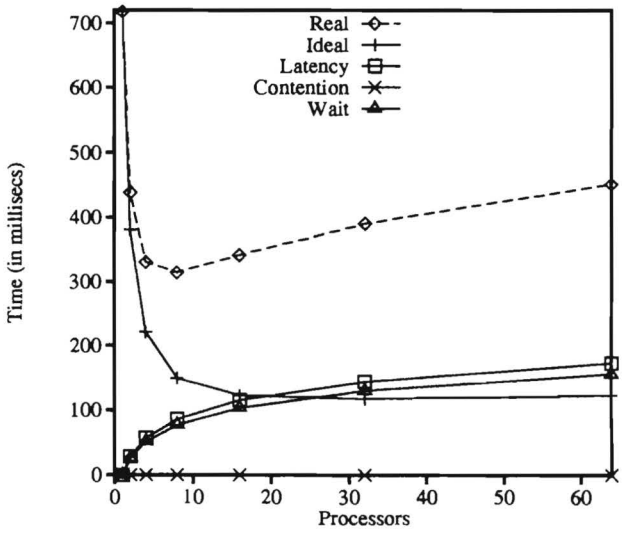

FIG. 16. IS: Overheads on cube.

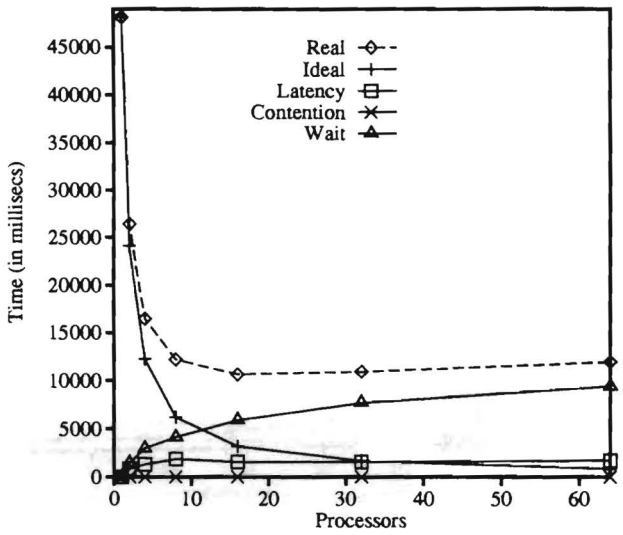

FIG. 17. CG: Overheads on cube.

putation among all the parallel processors. For EP, the overheads are marginal and the bulk of time is largely due to computation in the algorithm. For the other two kernels on the bus, contention becomes a bigger problem than latency with increasing numbers of processors, as explained earlier. For large numbers of processors, a considerable wait time is seen. The kernels consist of computation phases and communication phases. All the computation phases are load balanced among the processors and they arrive at a communication phase around the same time. The work-imbalance overhead (W_w) is mainly due to the logarithmic reduce operation where the number of processors participating is halved at each step. This intuitively suggests that most of the wait time is due to latency (W_l) and contention (W_c) incurred in previous messages. The measurements (see Tables IV and V) confirm this intuition. These measurements are for 64-node bus and cube systems for all three kernels.

Figures 12, 13, and 14 also show the relative impact of latency and contention overhead functions on performance. For smaller numbers of processors, latency is a more dominant factor than contention in limiting performance. But as mentioned earlier, the latency grows logarithmically (because of the structure of the algorithms) and is soon superseded by the faster-than-linear growing contention overhead function. This transition occurs at around 40 processors for IS and at around 12 processors

TABLE IV
Wait Times on the Bus

Kernel	W_w	W_l	W_c
EP	0.0%	100.0%	0.0%
IS	0.0%	31.1%	68.9%
CG	0.4%	34.4%	65.2%

TABLE V
Wait Times on the Cube

Kernel	W_w	W_l	W_c
EP	0.0%	100.0%	0.0%
IS	0.0%	100.0%	0.0%
CG	0.2%	99.8%	0.0%

for CG on the bus platform. Latency and contention overheads have very little effect on the performance of EP.

To understand the effect of varying the bus bandwidth on the contention overhead function, we simulate a 64-node bus platform for the three kernels and vary the cycle time of the bus. Figure 18 shows the result of this simulation. The overheads are given in seconds for CG, in milliseconds for IS, and in microseconds for EP. An interesting observation from this graph is that the contention overhead seen by a processor increases linearly with an increase in the bus cycle time. The contention overhead is affected the least for CG even though the net contention seen by a processor is the maximum of the three kernels (see Fig. 18). IS exhibits the maximum change in contention overhead while EP falls in between.

6.1. *Validation*

We validate our simulation by executing the kernels on comparable parallel machines and present sample validation results in this subsection. Figures 19, 20, and 21 compare the execution times for the EP, IS, and CG kernels, respectively, on an iPSC/860 and on SPASM simulating an iPSC/860. The curves are identical for EP while there is around a 10–15% deviation for CG and around a 15–20% deviation for IS. However, the shapes

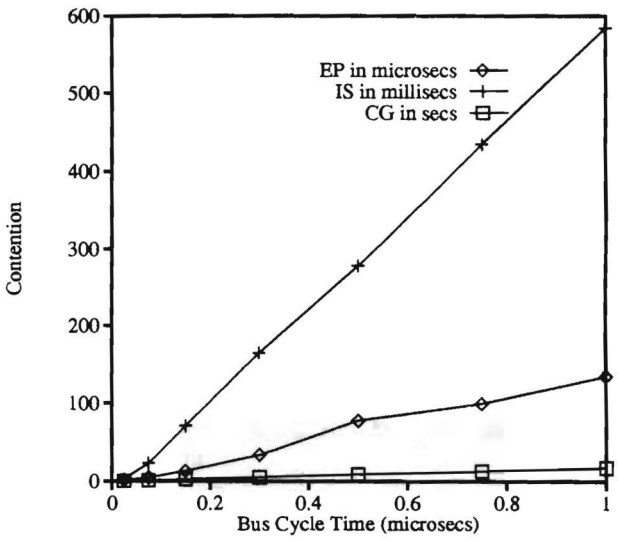

FIG. 18. Effect of bus cycle time on contention (64 PEs).

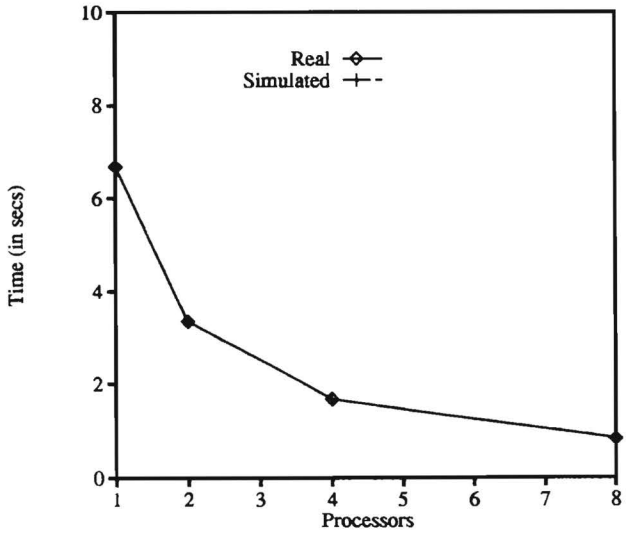

FIG. 19. EP on cube: Validation.

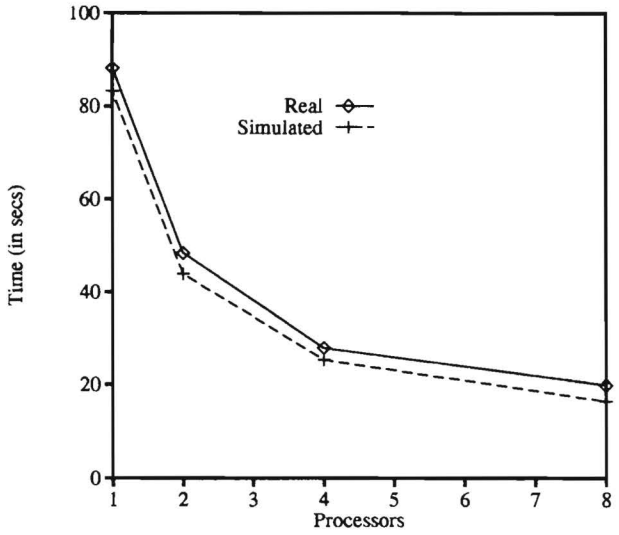

FIG. 21. CG on cube: Validation.

of the real and simulated curves are very similar, indicating that trends predicted by the simulation are accurate to within a constant factor. The deviation is largely due to inaccuracies in our estimation of the time taken for execution of the processor instruction streams. As mentioned in Section 3, we use the special (simulated) instructions to update the simulation clock of a processor for the instructions that are executed at the speed of the native processor. If we are to use UNIX system calls to measure this time interval, then we are limited by the least count of the UNIX timers. The least count of the UNIX timer calls on the SPARC station is in milliseconds and this can severely impact our measurements. Hence, we have resorted to calculating these time quanta manually and introducing the appropriate instrumentation code in our source programs. These manual measurements may have contributed to the inaccuracies in the estimation. We propose to use the augmentation technique used in other similar simulation studies [5, 7] to overcome these inaccuracies.

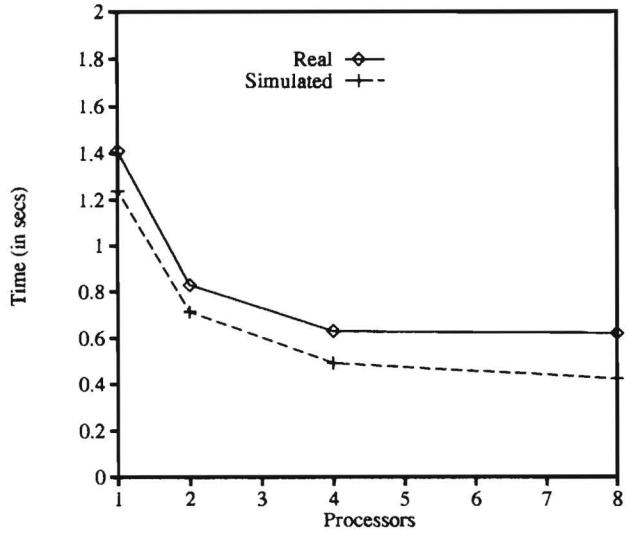

FIG. 20. IS on cube: Validation.

7. SCALABILITY IMPLICATIONS

In this section we illustrate how the top-down approach, in particular the overhead functions defined by the simulator, can be used for drawing conclusions regarding the scalability of a parallel system. It should be noted that these scalability projections are purely in the context of the specific algorithm–architecture pairs. For each of the three kernels, we discuss how the overhead functions are expected to grow with system size (both the number of processors as well as the problem size) based on the datapoints collected using the simulator. The computation time and the overhead functions are determined by interpolating these datapoints for each kernel. In reality, these functions are dependent on all aspects of a parallel system. However, for the sake of simplicity of analysis we express these as functions of the number of processors in the parallel system. The coefficients of these functions are constants for a given problem size and a given set of system parameters (such as link bandwidth).

7.1. EP Kernel

Table VI gives the computation time and the overhead functions for the EP kernel. Based on these functions, we can make the following observations.

- The computation time scales down linearly with the number of processors. In addition, it outweighs all the overheads and hence is the dominant factor in the execution time as can be seen from the coefficients associated with these functions.

TABLE VI
EP: Overhead Functions (in Milliseconds)

EP	BUS	Cube
Comp. time	$3873/p$	$3873/p$
Latency	$1.1 \log p$	$1.1 \log p$
Contention	$0.15 * 10^{-3}\lfloor p/2 \rfloor$	0
Wait	$\log p$	$\log p$

• Since the communication is confined to the logarithmic global sum operation, the latency overhead grows logarithmically with the number of processors. The coefficients for the latency overhead on the bus and cube are the same since EP uses small sized messages (4 bytes) for this operation.

• On the cube, there is no contention overhead for the logarithmic communication operation while it grows linearly with the number of processors on the bus. However, the associated coefficient is so small that its effect is negligible in absolute terms as well as relative to the latency overhead. The contention overhead becomes dominant compared to the latency only beyond several thousand processors.

• As is shown in Tables IV and V, the wait time is more dependent on the latency overhead than on the contention for this kernel, thus growing logarithmically with system size.

From these observations, we can conclude that the bus and cube systems scale well with the number of processors for the EP kernel. Even for a relatively small problem size (64 K in this case), it would take nearly 1000 processors for the overheads to start dominating. While an increase in the problem size would increase the coefficient associated with the computation time, it does not have any effect on the coefficients of the other overhead functions. Therefore the bus and cube systems scale well with the problem size as well for this kernel.

7.2. IS Kernel

Table VII gives the computation time and the overhead functions for the IS kernel. Based on these functions, we can make the following observations.

• In the ideal execution of the IS kernel, the computation time scales down logarithmically with the number of processors.

• As with EP, the latency overhead function is logarithmic for IS. The difference is that the coefficients for this function are different for the bus and the cube. Since the individual link bandwidths on the cube are lower than the bus bandwidth, the longer messages used by IS incur a larger latency on the cube.

• The logarithmic communication does not incur any contention on the cube. But on the bus, the contention grows quadratically with the number of processors and exceeds the latency overhead beyond 40 processors.

• The wait time on the cube is purely dependent on the latency overhead (Table V) and is thus logarithmic. On the other hand, the wait time on the bus depends on the latency and contention overheads (Table IV) and the resulting function thus lies in between these two overhead functions.

From the above observations, we can conclude that the bus and cube systems are not scalable with respect to the IS kernel. For the bus system, the contention function (being quadratic) starts dominating the computation time with increasing system size. On the cube, despite the lack of contention, the coefficient associated with the latency overhead is significant compared to that associated with the computation time and becomes the limiting factor with increasing number of processors.

An increase in the problem size will increase the coefficients associated with the computation time as well as the latency overhead (since the messages in IS are dependent on the problem size). For the bus system, Fig. 18 indicates that the contention increases linearly with the link latency. Therefore, increasing the problem size is likely to make both the cube and the bus systems less scalable for this kernel.

7.3. CG Kernel

Table VIII gives the computation time and the overhead functions for the CG kernel. Based on these functions, we can make the following observations.

• The computation time scales down linearly with the number of processors.

• As with the previous two kernels, the latency overhead is logarithmic in the number of processors.

• The contention overhead on the bus increases lin-

TABLE VII
IS: Overhead Functions (in Milliseconds)

IS	BUS	Cube
Comp. time	$717/(1 + \log p)$	$717/(1 + \log p)$
Latency	$2.5 \log p$	$28.9 \log p$
Contention	$0.01 * p^2$	0
Wait	$2.28 * \lfloor p/2 \rfloor$	$26 \log p$

TABLE VIII
CG: Overhead Functions (in Milliseconds)

CG	BUS	Cube
Comp. time	$48146/p$	$48146/p$
Latency	$65 \log p$	$500 \log p$
Contention	$45 * \lfloor p/2 \rfloor$	0
Wait	$75 * \lfloor p/2 \rfloor$	$1400 \log p$

early with the number of processors and becomes the limiting factor beyond 10 processors. On the cube, contention is nonexistent.

• The wait time on the cube is a logarithmic function (since there is no contention), while it is a linear function on the bus (since contention dominates latency in this case).

Even though the computation time for the CG kernel scales down linearly with the number of processors, the hardware overheads limit the scalability of this parallel system. However, increasing the problem size has a favorable impact on scalability of this kernel for both platforms. While the coefficient associated with the computation time increases quadratically with the problem size, those associated with latency and contention are likely to grow only linearly. Therefore, increasing the problem size is likely to make both the cube and the bus systems more scalable for this kernel.

8. CONCLUDING REMARKS

We have proposed a top-down approach to separate and quantify the different overheads in a parallel system that limit its scalability. We have used a combination of execution-driven simulation and experimentation to implement this approach. We use experimentation to understand the performance implications of real applications on real architectures and to identify interesting kernels occurring in such applications. The kernels are then used in our simulation to separate the different overheads that cause non-ideal behavior. We have developed a simulation platform (SPASM) to conduct this study. SPASM provides an elegant way of isolating the algorithmic overhead and interaction overhead in a parallel system and further separating them into their respective components. We illustrated our approach by simulating the NAS parallel kernels on bus-based and hypercube-based message-passing platforms on SPASM and isolated the algorithmic effects such as serial and work-balance overheads and the interaction effects such as latency and contention.

9. FUTURE WORK

Currently, we are limited by the resource constraints inherent in sequential simulation for simulating large parallel systems. We are exploring the viability of both conservative and optimistic methods of parallel simulation on different hardware platforms for overcoming this limitation. There are several interesting directions for extending this work. One is to identify and quantify other overheads in a parallel system such as scheduling, synchronization, and caching. Another direction is to include a wider range of hardware platforms and a broader application domain.

ACKNOWLEDGMENTS

The authors thank the anonymous referees for their comments, which helped put the results of this work in the proper perspective in addition to improving the quality of the presentation.

REFERENCES

1. Agarwal, A. Limits on interconnection network performance. *IEEE Trans. Parallel Distrib. Systems* **2,** 4 (Oct. 1991), 398–412.
2. Amdahl, G. M. Validity of the single processor approach to achieving large scale computing capabilities. In *Proceedings of the AFIPS Spring Joint Computer Conference*, Apr. 1967, pp. 483–485.
3. Anderson, T. E. The performance of spin lock alternatives for shared-memory multiprocessors. *IEEE Trans. Parallel Distrib. Systems* **1,** 1 (Jan. 1990), 6–16.
4. Bailey, D. *et al.* The NAS parallel benchmarks. *Internt. J. Supercomputer Appl.* **5** 3 (1991), 63–73.
5. Brewer, E. A., Dellarocas, C. N., Colbrook, A., and Weihl, W. E. PROTEUS: A high-performance parallel-architecture simulator. Tech. Rep. MIT-LCS-TR-516, Massachusetts Institute of Technology, Cambridge, MA, 02139, Sept. 1991.
6. Chen, D., Su, H., and Yew, P. The impact of synchronization and granularity on parallel systems. In *Proceedings of the 17th Annual International Symposium on Computer Architecture*, 1990, pp. 239–248.
7. Covington, R. G., Madala, S., Mehta, V., Jump, J. R., and Sinclair, J. B. The Rice parallel processing testbed. In *Proceedings of the ACM SIGMETRICS 1988 Conference on Measurement and Modeling of Computer Systems*, Santa Fe, NM, May 1988, pp. 4–11.
8. Cvetanovic, Z. The effects of problem partitioning, allocation, and granularity on the performance of multiple-processor systems. *IEEE Trans. Comput. Systems* **36,** 4 (Apr. 1987), 421–432.
9. Cypher, R., Ho, A., Konstantinidou, S., and Messina, P. Architectural requirements of parallel scientific applications with explicit communication. In *Proceedings of the 20th Annual International Symposium on Computer Architecture*, May 1993, pp. 2–13.
10. Eggers, S. J., and Katz, R. H. The effect of sharing on the cache and bus performance of parallel programs. In *Proceedings of the Third International Conference on Architectural Support for Programming Languages and Operating Systems*, Boston, MA, April 1989, pp. 257–270.
11. Gustafson, J. L., Montry, G. R., and Benner, R. E. Development of parallel methods for a 1024-node hypercube. *SIAM J. Scientific Statist. Comput.* **9,** 4 (1988), 609–638.
12. Intel Corp., *Intel iPSC/2 and iPSC/860 User's Guide*, 1989.
13. Jamieson, L. H. Characterizing parallel algorithms. In Jamieson, L. H., Gannon, D. B., and Douglas, R. J. (Eds.). *The Characteristics of Parallel Algorithms*. MIT PRESS, Cambridge, MA, 1987, pp. 65–100.
14. Karp, A. H., and Flatt, H. P. Measuring parallel processor performance. *Comm. ACM* **33,** 5 (May 1990), 539–543.
15. Kumar, V., and Rao, V. N. Parallel depth-first search. *Internat. J. Parallel Programming* **16,** 6 (1987), 501–519.
16. Kung, H. T. The structure of parallel algorithms. In Yovits, M. C. (Ed.). *Advances in Computers* Academic Press, New York, 1980, Vol. 19, pp. 65–112.
17. Madala, S., and Sinclair, J. B. Performance of synchronous parallel algorithms with regular structures. *IEEE Trans. Parallel Distrib. Systems* **2,** 1 (Jan. 1991), 105–116.
18. McMahon, F. H. The Livermore Fortran kernels: A computer test of the numerical performance range. Tech. Rep. UCRL-53745, Lawrence Livermore National Laboratory, Livermore, CA, Dec. 1986.

19. Mellor-Crummey, J. M., and Scott, M. L. Algorithms for scalable synchronization on shared-memory multiprocessors. *ACM Trans. Computer Systems* **9,** 1 (Feb. 1991), 21–65.

20. Patel, J. H. Analysis of multiprocessors with private cache memories. *IEEE Trans. Computer Systems* **31,** 4 (Apr. 1982), 296–304.

21. Pfister, G. F., and Norton, V. A. Hot spot contention and combining in multistage interconnection networks. *IEEE Trans. Computer Systems*, **C-34,** 10 (Oct. 1985), 943–948.

22. Reinhardt, S. K., *et al.* The Wisconsin wind tunnel: Virtual prototyping of parallel computers. In *Proceedings of the ACM SIGMETRICS 1993 Conference on Measurement and Modeling of Computer Systems*, Santa Clara, CA, May 1993, pp. 48–60.

23. Rothberg, E., Singh, J. P., and Gupta, A. Working sets, cache sizes and node granularity issues for large-scale multiprocessors. In *Proceedings of the 20th Annual International Symposium on Computer Architecture*, May 1993, pp. 14–25.

24. Sivasubramaniam, A., Ramachandran, U., and Venkateswaran, H. Message-passing: Computational model, programming paradigm, and experimental studies. Tech. Rep. GIT-CC-91/11, College of Computing, Georgia Institute of Technology, Feb. 1991.

25. Sivasubramaniam, A., Shah, G., Lee, J., Ramachandran, U., and Venkateswaran, H. Experimental evaluation of algorithmic performance on two shared memory multiprocessors. In Suzuki, N. (Ed.). *Shared Memory Multiprocessing*. MIT Press, 1992, pp. 81–107.

26. Sivasubramaniam, A., Singla, A., Ramachandran, U., and Venkateswaran, H. Machine abstractions and locality issues in studying parallel systems. Tech. Rep. GIT-CC-93/63, College of Computing, Georgia Institute of Technology, Oct. 1993.

27. Sivasubramaniam, A., Singla, A., Ramachandran, U., and Venkateswaran, H. An approach to scalability study of shared memory parallel systems. In *Proceedings of the ACM SIGMETRICS 1994 Conference on Measurement and Modeling of Computer Systems*, May 1994, pp. 171–180.

28. Sun, X-H., and Gustafson, J. L. Towards a better parallel performance metric. *Parallel Computing* **17** (1991), 1093–1109.

29. Vrsalovic, D. F., Siewiorek, D. P., Segall, Z. Z., and Gehringer, E. Performance prediction and calibration for a class of multiprocessors. *IEEE Trans. Computer Systems* **37,** 11 (Nov. 1988), 1353–1365.

30. Wyllie, J. C. *The complexity of parallel computations*. Ph.D. thesis, Department of Computer Science, Cornell University, Ithaca, NY, 1979.

Received May 1, 1993; revised February 11, 1994; accepted March 28, 1994

ANAND SIVASUBRAMANIAM received his B. Tech. in Computer Science from the Indian Institute of Technology, Madras, in 1989, and an M.S. in Computer Science from the Georgia Institute of Technology in 1991. Since 1989 he has been a Ph.D. student in the College of Computing at the Georgia Institute of Technology, where he holds a Graduate Research Assistantship. His research interests are in computer architecture, parallel processing, simulation and evaluation of computer systems, and operating systems. His research has focused on studying the synergy between parallel algorithms and architectures, using experimentation, simulation, and modeling, toward an understanding of the scalability of large-scale parallel systems.

AMAN SINGLA received hs B.Tech. in Computer Science and Engineering from the Indian Institute of Technology, Delhi, in 1992. He is currently a Ph.D. student and a graduate research assistant at the Georgia Institute of Technology. His interests include parallel processing, computer architecture, distributed systems, and approximation algorithms.

UMAKISHORE RAMACHANDRAN received his Ph.D. in Computer Science from the University of Wisconsin, Madison, in 1986, and is currently an Associate Professor in the College of Computing at the Georgia Institute of Technology. He has researched extensively in the architectural design, programming, and analysis of parallel and distributed systems. He was a co-principal investigator for the Clouds distributed operating system project at Georgia Tech. His current interests are in investigating software and hardware mechanisms for building scalable shared memory systems and studying the scalability of parallel systems from an applications perspective. He was the recipient of a Presdential Young Investigator (PYI) Award from the National Science Foundation (NSF) in 1990 and the Georgia Tech doctoral thesis advisor award in 1993.

H. VENKATESWARAN received his Ph.D. in Computer Science from the University of Washington in 1986, and is currently an Associate Professor in the College of Computing at the Georgia Institute of Technology. His primary interests relate to the theoretical issues in parallel computation. He is also concerned with practical issues that affect the fundamental assumptions in such theoretical studies. He is a co-principal investigator in a project that studies the impact of architectural features on the performance of parallel algorithms.

JOURNAL OF PARALLEL AND DISTRIBUTED COMPUTING 21, 323–333 (1994)

Evaluating Multigauge Architectures for Computer Vision*

W. B. LIGON III†‡ AND U. RAMACHANDRAN§‖

†*Department of Electrical and Computer Engineering, Clemson University, Clemson, South Carolina 29634-0915; and* §*College of Computing, Georgia Institute of Technology, Atlanta, Georgia 30332-0280*

Heterogeneous computing is a technique for achieving high performance by providing a variety of different architectures to meet the needs of systems that are composed of tasks with widely different characteristics. Essential to the construction of heterogeneous systems is an understanding of the match between architecture and software and how that match can be used in deciding how to utilize the available computing resources. We present a theoretical framework, the PCI model, which defines corresponding characteristics of parallel programs and parallel architectures and defines the performance relationship between them in terms of these characteristics. We have encapsulated the concepts of the PCI model into RAW, a simulation environment that facilitates experimentation with the program/architecture relationship in terms of the PCI model. Using RAW, we have applied the PCI model to study the use of processor reconfigurable architectures (a type of heterogeneous system) in the context of computer vision applications. We present experimental results that demonstrate that these architectures perform better than static homogeneous architectures for such applications. © 1994 Academic Press, Inc.

1. INTRODUCTION

A heterogeneous computing system provides multiple architectures so that the needs of each phase or task of a program can be met to the fullest extent possible. This requires decomposing the target program into its component tasks, assigning each task to an architecture, parallelizing the tasks to run on that architecture, and providing the necessary coordination to deal with the partitioned program. One approach to realizing a heterogeneous system is to interconnect several different kinds of architectures via a network. This approach is appealing because it is relatively simple to realize with currently available commercial computers. Another approach to the development of heterogeneous computing systems is the use of reconfigurable architectures. These architectures can dynamically alter their logical structure to provide the desired mix of computational resources and computing modes. This approach allows resource allocation to be performed at run time in order to best suit the current state of the application.

Regardless of the approach used to develop heterogeneous systems, there are a number of issues one must address in their design. First, for each phase in the target application, one would like to know the best architecture possible for that phase. Second, one must decide how to interconnect the architectures in order to minimize the costs of coordinating the system such as communication and synchronization delays. To address both of these issues, one needs a quantitative measure of each phase's expected performance for any given architecture as compared to another; and a quantitative understanding of communication and synchronization costs in order to perform a cost/benefit analysis. The goal of our research is to address this problem by establishing a model for reasoning about the performance of a parallel program relative to a specific parallel architecture. Furthermore, we have developed an execution-driven simulation testbed that embodies this model called the *Reconfigurable Architecture Workbench* (RAW) [4], with which we can empirically explore the program/architecture relationship.

In Section 2 we present the *Processor, Control, Interconnection* (PCI) model for programs and architectures. The PCI model separates processor design, control structure, and interconnection issue and also accounts for heterogeneous and reconfigurable architectures. We further discuss the performance relationship of programs and architectures within the context of this framework. The PCI model is the primary contribution of our work because it provides a precise means for evaluating the performance benefit to be gained in utilizing a heterogeneous architecture for a specific application. In Section 3 we present a case study that demonstrates the use of RAW, which embodies the PCI model, in evaluating one type of heterogeneity for computer vision applications. Using RAW, we perform a quantitative evaluation of an application domain that intuition tells us should benefit from heterogeneous computing. The thesis of our work is that analysis of this type is needed to assess the benefits of heterogeneous computing.

2. THE PCI MODEL

The PCI model defines parallel programs as being composed of three components: (1) independent *instruction*

* This work is funded in part by an NSF PYI Award MIP-9058430.
‡ E-mail: walt@eng.clemson.edu.
‖ E-mail: rama@cc.gatech.edu.

0743-7315/94 $6.00

streams each of which execute in a sequential manner; (2) independent *data streams,* each of which execute instructions on different data objects; and (3) *communication,* which includes synchronization. PCI defines parallel architectures as being composed of three similar components: (1) *control units* (CUs) that process instruction streams; (2) *processing elements* (PEs) that execute the instructions issued by a given control unit; and (3) *interconnection networks* (ICNs) that allow communication and synchronization. We call these components the control configuration, the processor configuration, and the interconnection configuration. Finally, PCI defines three classes of reconfigurability: (1) processor reconfiguration allows the architecture to trade off the number of PEs for the precision and speed of the PEs; (2) control reconfiguration allows the assignment of PEs to control units to be changed; and (3) interconnection reconfiguration allows the communication capabilities of the parallel architecture to be modified.

2.1. The PCI Model for Parallel Programs

A parallel program consists of a set of data objects and a set of instructions for manipulating those objects. When a program is executed, it produces one or more instruction streams which are loosely defined as independent sequences of instructions issued when the program is executed. If a program's execution yields more than one instruction stream, the instructions of the various instruction streams may be issued simultaneously (parallel execution), or they may be interleaved in time (concurrent execution). Programs that define multiple instruction streams are termed *function parallel* or *control parallel* programs. Each instruction of an instruction stream carries out a specific manipulation on one or more data objects defined by the parallel program. A *data parallel* program defines a parallel data space *PD,* composed of n constituent data spaces pd_i. A parallel data object *PO* consists of n constituent objects po_i each of which exists in the corresponding data space pd_i (see Fig. 1). Parallel data objects may be manipulated simultaneously by a single instruction. Similarly, there is a singular data space *SD* where singular data objects are defined. This data space is further partitioned into a local data space, private to a given instruction stream, and a global data space *GD,* shared by all instruction streams.

Communication is the process of transferring information from one data space to another. Communication within an instruction stream is essentially data transfer between parallel data spaces. Communication between different instruction streams may transfer either data or control information. Communication generally takes one of two forms: either the sending and receiving messages, or access to data objects in the global data space. Synchronization is a form of communication that involves a class of instructions called WAIT instructions. A WAIT instruction is an instruction that does not complete until some specified event has occurred. These events are often the result of some type of communication. For example, an instruction stream may issue a *blocking receive* instruction (one type of WAIT) which does not complete until a message is received from some other instruction stream. Another important example is a *memory load* instruction which does not complete until the memory value is returned. Different systems may implement a variety of WAIT instructions, but in each case, the effect of these instructions is that a delay is introduced into the instruction stream's execution time which may be dependent on events that occur in other instruction streams.

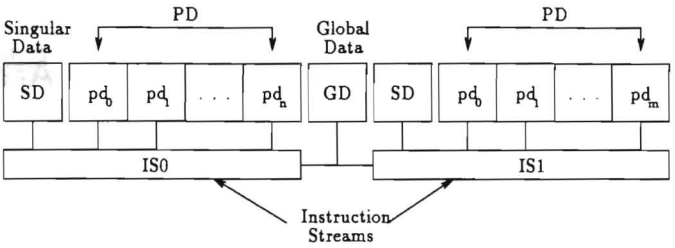

FIG. 1. PCI program model.

2.2. The PCI Model for Parallel Architectures

A parallel architecture consists of control units, processing elements, and communication channels (see Fig. 2). A CU consists of a program counter, instruction memory, and logic needed to generate control signals for the PEs. Branching and WAIT instructions are processed directly by the CUs, while all other instructions are handled by PEs. CUs fetch instructions and issue them in parallel to all PEs controlled by the CU. CUs may contain additional features that support the processing of multiple concurrent instruction streams, but only one instruction stream can be active during any one CU machine cycle.

PEs consist of execution logic, operand address generation logic, and memories that define the various data spaces. There always exists at least one PE for each control unit called the *singular* PE (labeled "PES" in the figure). This PE's memory defines the local data space and its execution logic carries out the instructions directed at the objects in either that data space or the global data space. In addition to the singular PE, each control

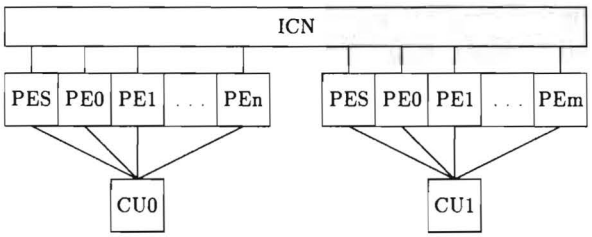

FIG. 2. PCI architecture model.

unit may have 0 ··· *n parallel* PEs (labeled "PE0" to "PE7" in the figure) which define and operate on objects defined in parallel data spaces. A single PE may process several constituent data spaces of a parallel data space. A PE is characterized by its *word width*, which defines the largest data object that can be processed in a single machine cycle; and *instruction timings*, which are the number of machine cycles required to execute each of the instructions in the instruction set.

ICNs consist of two basic elements: *links* and *switches*. A link is a medium that is used to pass information from node to node. A switch is an active device in a network that serves to pass information from the end of one link to the beginning of another. A switch may make routing decisions, it may store messages for later transmission, and it may choose to allocate or deallocate network resources. ICNs are characterized by a set of parameters: *topology* describes the collection of links and switches in the ICN and their interconnection; *bandwidth* describes the throughput of the links and switches; *latency* describes the delay incurred through a link or a switch by a flit[1]; and *switch capabilities* describe the level of intelligence incorporated into a switch such as routing logic, queueing, and policies.

2.2.1. Heterogeneous and Reconfigurable Architectures

Heterogeneous architectures are those that are composed of two or more different architectures. The distinction may be with respect to any of the three dimensions. For example, an SIMD machine connected to an MIMD machine by an ICN is one type of heterogeneous architecture; a fine grain machine connected to a coarse grain machine by a LAN is another. An example is the Image Understanding Architecture [12]. The PCI model does not require that every PE, CU, or ICN be identical, so modeling heterogeneous architectures with PCI is straightforward. A special type of heterogeneity is architectural reconfigurability. Whereas a heterogeneous system usually refers to a static collection of architectures, architectural reconfiguration is the process of altering the logical view of the machine via special interconnection hardware in order to give the appearance of a different architecture. Reconfigurable architectures may appear at any one time to be homogeneous but reflect distinct configurations over the lifetime of the parallel program. This type of architecture is *temporally* heterogeneous. Other reconfigurable architectures can achieve configurations that are themselves heterogeneous [8, 9]. The main advantage of reconfigurable architectures is that they provide a means for allocating resources in order to yield the most efficient computational engine over a range of possible situations, rather than requiring a static configuration. This benefit is at the cost of complex custom hardware, and the inability to optimize for a single logical organization. Reconfigurable architectures exhibit three types of reconfigurability.

Interconnection reconfigurability describes the flexibility of an ICN in allocating its resources to provide service between two terminals. Nonreconfigurable ICNs have only a fixed set of resources for building a path between terminals. A reconfigurable network allocates switches and links from a pool to build a path between terminals. Essentially, reconfigurability is a means of providing a high degree of service for short periods of time over a subset of the system with fewer total resources. CHiP [10] is a representative architecture.

Control reconfigurability (also known as multimode capability) describes the ability of an architecture to partition the PEs in the system among the CUs in various ways. Control reconfigurable machines can be configured as MIMD, SIMD, and Multiple-SIMD systems. Examples include CM-2 [2], TRAC [8], and PASM [9].

Processor reconfigurability (also known as multigauge capability) is the ability of the architecture to trade off faster and/or higher precision PEs for more numerous PEs. *Precision reconfiguration* allows low-precision PEs to be joined to form higher-precision PEs. Data used in multiple configurations must be reformatted during reconfiguration, thus incurring potentially significant overhead unless special hardware is provided to allow access from any configuration. This type of processor reconfiguration is provided in TRAC and DCG [3]. *Capability reconfiguration* allows full-precision boolean logic units to be combined to support more complex arithmetic operations such as multiplication and floating point operations [6]. In such systems the CU utilizes a microsequencer to decode complex arithmetic operations into a sequence of microinstructions. In order to perform capability reconfiguration on the PEs, the CU executes a control-parallel microprogram on interleaved sets of the simple PEs. For example, a multiplier can be constructed by using one simple PE as a shifter, and an adjacent simple PE as an adder to perform shift-add multiplication considerably faster than using one simple PE. To implement this capability, the CU must provide multiple control busses to the PEs, and multiple control stores, one for each bus (see Fig. 3). Routines for simple PEs have the same code loaded at corresponding addresses in the various control stores. Routines for the complex PEs have different code loaded at the same address. In addition, there must exist some means for passing data between the simple PEs and performing basic coordination tasks.

2.3. Understanding the Performance of Programs Relative to Architectures

We now consider how the relationship between these models results in a given level of machine performance. Our measure of performance is the execution time of the program on a given architecture. The execution time of a

[1] A *flit* is the number of bits that can be transmitted in a single network clock cycle.

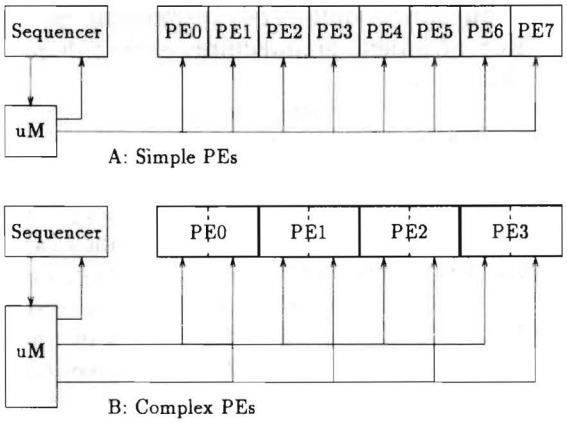

FIG. 3. Capability reconfiguration.

program on a particular architecture can be found by considering the instructions issued by each CU. The execution time of m instruction streams running concurrently on a single CU is:

$$T_{\text{execution}}(P) = \sum_{k=0}^{m} \sum_{j=0}^{n(k)} t_{\text{instruction}}(i_{j(k)}),$$

where $n(k)$ is the number of instructions in instruction stream I_k. The function $t_{\text{instruction}}(i_{j(k)})$ is the instruction time which is defined as

$$t_{\text{instruction}}(i_{j(k)}) = t_{\text{issue}}(i_{\text{next}}) - t_{\text{issue}}(i_{j(k)}),$$

where $i_{j(k)}$ is instruction i_j of instruction stream I_k and i_{next} is the next instruction issued by the CU that instruction stream I_k is executing on. In the case where I_k is the only instruction stream executing on the CU, i_{next} is guaranteed to be $i_{j(k)+1}$, but this may or may not be the case otherwise.

In the case where a program's m instruction streams are executing in parallel on the target architecture, each on a different CU the execution time of the program is the same as that of the longest instruction stream:

$$T_{\text{execution}}(P) = \max(T_{\text{execution}}(I_k))k = 0 \cdots m.$$

2.3.1. *The Effect of Control Units*

Control units in excess of the number of instruction streams have no effect on the program's performance whatsoever. If a program has multiple instruction streams and they all execute concurrently on a single CU, the CU can process instructions from other instruction streams during the time one instruction stream is waiting, thus WAIT instructions have relatively little effect on the overall execution time of the program. As more and more CUs are made available to process instruction streams in parallel, the execution time of the program is generally reduced because each CU has fewer instructions to process. However, as more parallel execution is employed, the instruction time of the WAIT instructions may increase. If this occurs, then the benefits of parallel execution are diminished. Considering the effect of taking a program P with two instruction streams I_0 and I_1. Assume the execution time of P using one CU is $T_{\text{sequential}}(P)$ and the contribution to the execution time of P by I_0 is $T_{\text{sequential}}(I_0)$ and similarly $T_{\text{sequential}}(I_1)$ for I_1. Now, the execution time of I_0 and I_1 using two CUs is

$$T_{\text{parallel}}(I_0) = T_{\text{sequential}}(I_0) + \Delta T_{\text{WAIT}}(I_0)$$
$$T_{\text{parallel}}(I_1) = T_{\text{sequential}}(I_1) + \Delta T_{\text{WAIT}}(I_1)$$

and the execution time of P is

$$T_{\text{parallel}}(P) = \max(T_{\text{parallel}}(I_0), T_{\text{parallel}}(I_1)).$$

$T_{\text{WAIT}}(I_k)$ is the sum of the instruction execution time $t(i_W)$ of all WAIT instructions i_W, each of which can be decomposed into the sum of two values: $t_{\text{sync}}(i_W)$ and $t_{\text{comm}}(i_W)$. The synchronization time of a WAIT instruction $t_{\text{sync}}(i_W)$ is the difference between the time of the actual occurrence of the event the instruction is waiting for and the issue time of the WAIT instruction. t_{sync} is an abstract value that may be positive or negative, depending on whether an event occurs before or after the issue of the WAIT instruction. The communication time of a WAIT instruction $t_{\text{comm}})(i_W)$ is the delay between the actual occurrence of an event and the time the CU receives notice of the event due to message latency. t_{comm} is a physical value and thus is strictly non-negative. With these definitions we can find the delay introduced by a WAIT instruction as

$$t(i_W) = \max(t_{\text{sync}}(i_W) + t_{\text{comm}}(i_W), 0)$$

which is also strictly nonnegative.

When there are many instruction streams executing together on a single CU, the effect of WAIT instructions is minimized by delaying their issue and processing other instruction streams instead. Ideally, this creates negative synchronization time and thus the WAIT instructions have little effect. During parallel execution, $T_{\text{WAIT}}(I)$ may increase relative to the concurrent execution for two reasons. First, even though there may still be multiple instruction streams on a CU, there may be times when all of them need to issue a WAIT and thus there are no other instructions to issue. Second, by definition the fact that the event of interest may be occurring on a remote CU, an increase in t_{comm} may be experienced.

2.3.2. *The Effect of Processing Elements*

The time required by the processing elements to perform the manipulations specified by an instruction i on d data streams with p PEs is:

$$t_{\text{parallel}}(i) = t_{\text{sequential}}(i)\lceil d/p \rceil.$$

The natural conclusion is that faster PEs produce a faster machine, and more PEs produce a faster machine until there are more PEs than data streams. In designing an architecture, one has to contend with several real-life limits, such as the amount of hardware that can be made available as PEs based on an acceptable cost for the machine, and physical constraints such as power, cooling, size, and signal propagation delays. Given these limits, one must choose the best trade-off between the number of PEs and the speed of the PEs.

2.3.3. *The Effect of Interconnection Networks*

The effect of ICNs on program performance is in the communication component of the WAIT instruction. The quantity of interest in measuring communication time is message latency which is the difference in time between the sending of the first flit in the message and the receiving of the last flit in the message. This quantity depends on the parameters of the ICN. The relationship between these parameters is beyond the scope of this paper, and the reader is referred to texts such as [1]. One should note that one potentially large component of this equation is that of resource contention which can be highly dependent on the behavior of the parallel program.

2.3.4. *The Effect of Heterogeneity*

Heterogeneous architectures exploit the different characteristics of the various phases and tasks of a parallel program. The basic premise is this: if one phase of a program performs better on architecture x than architecture y, and a different phase of the same program performs better on y than x, then the program should perform better on a heterogeneous architecture xy that is composed of both x and y than on either x or y alone. Whether or not this is actually true depends on how much better each phase runs on its preferred architecture, and the overhead OH associated with the heterogeneous configuration. For a program P composed of phases G and H, the execution time of P on x is

$$T_x(P) = T_x(G) + T_x(H)$$

and on y is

$$T_y(P) = T_y(G) + T_y(H)$$

If $T(G_x) > T(G_y)$ and $T(H_y) > T(H_x)$, then the performance improvement obtained when run on heterogeneous architecture xy is

$$\Delta T_{xy,y}(P) = \Delta T_{x,y}(H) - OH$$

and

$$\Delta T_{xy,x}(P) = \Delta T_{y,x}(H) - OH,$$

where $\Delta T_{i,j}(P)$ is the performance improvement for P when run on architecture i over architecture j. This assumes architectures x, y, and xy provide roughly the same amount of hardware, thus parallelism that may exist between G and H has already been accounted for in x and y.

Why would one program perform better on one architecture than another? Potential reasons are: (1) The program cannot utilize the resources provided by an architecture; or (2) a different logical configuration of resources would provide improved economies of scale. If the architecture provides more PEs than the program has data streams or more CUs than the program has instruction streams, then clearly resources remain idle. Similarly, if the architecture defines special purpose hardware such as a floating point unit, and the computation is primarily integer, then again, those resources cannot be utilized. The second program is more subtle. More complex computational units are, at least in theory, less efficient than simple computational units because there is overhead involved in coordinating the added complexity. For example, an n-bit PE is not necessarily n times faster than a 1-bit PE, in part because the carry computation increases the critical path of the adder circuits. Our research does not seek to outline the exact performance relationship of various designs. Rather, we note that each specific hardware configuration has its own performance characteristics, and that a comparison between them requires an empirical analysis of the cost/benefit function. It is this analysis that concerns us.

It can be seen in the preceding discussion that the ability of an architecture to execute a parallel program efficiently is highly dependent on the program's structure, which in many cases is dynamic. Thus, our model cannot give us an analytical evaluation of the utility of a given architecture, rather it provides a basis for an empirical evaluation. The PCI model is general and applicable to heterogeneous as well as reconfigurable architectures. We have incorporated the PCI model in an empirical tool called the Reconfigurable Architecture Workbench (RAW). RAW is a simulation platform that allows studying applications in the context of parallel architectural parameters. Details of the simulator may be found in [4]. In the next section, we present a case study that focusses on using RAW to evaluate multigauge architectures for computer vision applications. We have selected computer vision as our target application because there is a significant amount of parallelism and intuitively, the applications lend themselves to the use of heterogeneous architectures. This intuition is gleaned from the observation that vision systems span several levels of abstraction, each of which typically requires different data representations and algorithms for manipulating the objects at that level. The goal of the study is to quantify the intuitive benefit of multigauge capability inherent in this application.

TABLE I
Programs and Phases Used in the Experiments

Program/Phase	Description
Filter	
Communicate 1	Trade border values with neighbors
Median	Median filter on image partition
Fix Corners	Handle image borders
Communicate 2	Trade border values with neighbors
Sobel	Sobel transform on image partition
Label	
Initialize	
Build Borders	Determine pixels on region borders
Redistribute	Load balance border pixels
Number Regions 1	Distance doubling to find unique ID
Merge Borders	Handle nested regions
Number Regions 2	Distance doubling to find unique ID
Fill Regions	Propagate region ID throughout region
Corners	
Unpack	Data decoding
k-Curvature	Check region border curvature at distance K
Smoothing	Smooth curvatures with gaussian
Zero Crossings	Find zero crossings in curvature
Mark Corners	Corners have high curvature and zero crossing
Rectangles	
Initialize	
Eliminate Nodes	Eliminate noncorner border pixels
Scan for Corners 1	Eliminate regions with <3 corners
Convex Hull	Eliminate corners not on the hull
Scan for Corners 2	Eliminate regions with <3 corners
Find Right Angles	Measure corner angles
Find Rectangles	Rectangles have 3 consecutive right angles
Build Rectangle DB	Extract rectangle parameters

3. A STUDY OF MULTIGAUGE ARCHITECTURES

The algorithms used in our study are drawn from the DARPA Image Understanding Benchmark [11]. Table I summarizes these algorithms. The algorithms are hand parallelized and provide the input to RAW. Details of the parallel implementation may be found in [4].

Our experiments utilize three different PE models, referred to as *Config* 1, *Config* 2, and *Config* 3. All of the PE models are based on the bit-serial design in Hillis [2]. Config 1 is composed of a single PE of this design, while Config 2 and Config 3 are composed of multiple instances of this PE bound with reconfiguration hardware to allow both precision and capability reconfiguration including a carry-lookahead circuit, a barrel-shifter, and a reconfigurable memory module such as those outlined in [5]. Timing for the PEs does not account for carry computation used in Config 2 and Config 3, thus extra machine cycles are needed to allow for the delay of these circuits. The reconfigurable memory modules allow reformatting costs to be avoided for up to 1MB of memory per Config 3 PE. Table II summarizes the PE configurations including the number of 1-bit PEs used in the implementation and the number of logical PEs available to the program. Note that the configurations we simulate utilize a constant amount of hardware.

TABLE II
PE Configurations Used in the Experiments

PE Configuration	Word width	Multiply/FP support	Number of units per logical PE	Number of logical PEs
Config 1	1	N	1	16K
Config 2	32	N	32	512
Config 3	32	Y	64	256

All of the architectures studied are SIMD machines. Exactly one instruction stream executes on a given CU, so context switching, scheduling, etc. are not an issue. The ICN model is a "*k*-ary *n*-cube" configured as a binary hypercube with serial links. The details of this model can be found in [7] and is not relevant to understanding the scope of this paper.

3.1. Experimental Method

In [5] we outline an experimental methodology for utilizing RAW in the design of parallel architectures. In this paper we utilize the same methodology. We first study the performance of each of the four programs on an increasing number of Config 3 PEs, noting any change in the performance characteristic. Based on the results of this initial experiment, we select three distinct configurations of the processing elements (as outlined in Table II.) and simulate the program in each configuration. Next we study the effects of processor reconfiguration by running each program using the configuration that provides the best performance. Finally, we repeat the last experiment, this time considering the various phases of each program and running each phase in the configuration that provides the best performance. In each case we include the costs of coordinating the various configurations due to architectural reconfiguration.

3.2. Experimental Results

Figures 4A and 4B show the total execution time and execution profile of the filter program when executed on 1 to 16K Config 3 PEs. The first graph indicates that the program is characterized by a large amount of usable parallelism. The profile shows the parallelism to be primarily due to the median and Sobel phases. Figures 4C and 4D indicate that PE model 1 performs the best for the median and Sobel phases by a factor of nearly 3, and all configurations are roughly equal for the communication phases.

Figures 5A and 5B show the total execution time and profile for the `label` program for 1 to 16K Config 3 PEs. These graphs indicate that the `label` program does not behave as well as the `filter` program. While the `filter`

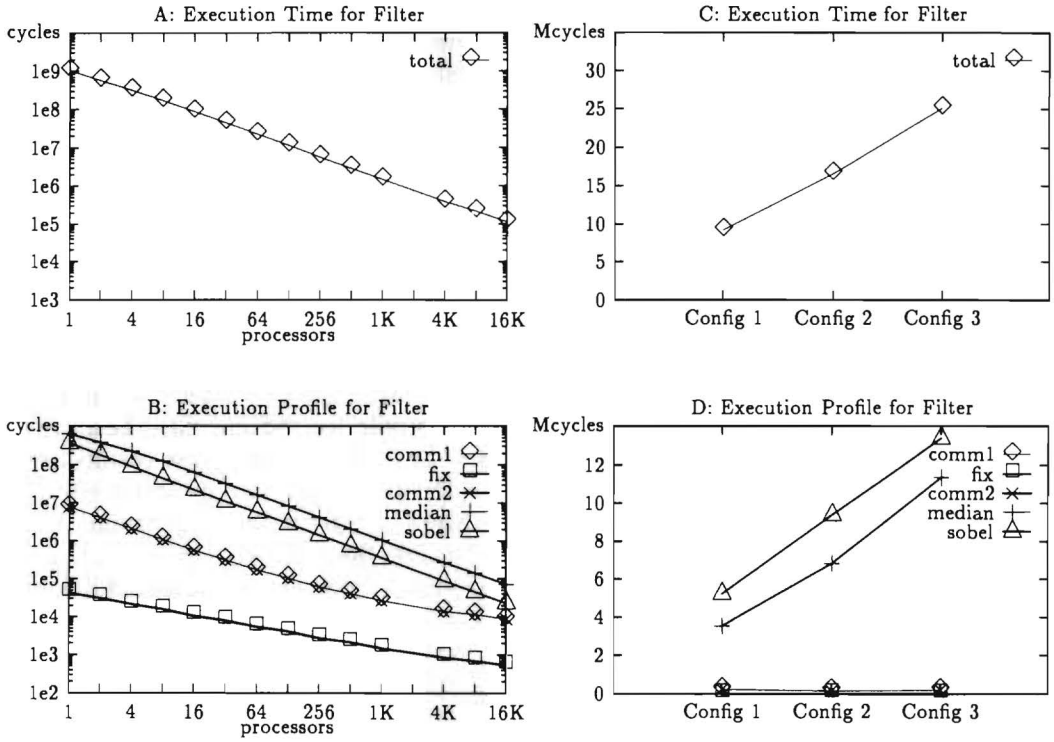

FIG. 4. Execution time and profile for the `filter` program.

program achieved a speedup of nearly four orders of magnitude for 16K PEs, the `label` program achieves less than 3 orders of magnitude speedup. Also, the graph for the `label` program curves up sharply between 4 and 16K PEs. The profile shows that this is primarily due to the number region phase. The rest of the phases show a speedup similar to that of the overall curve, except the initialization step, which shows over four orders of mag-

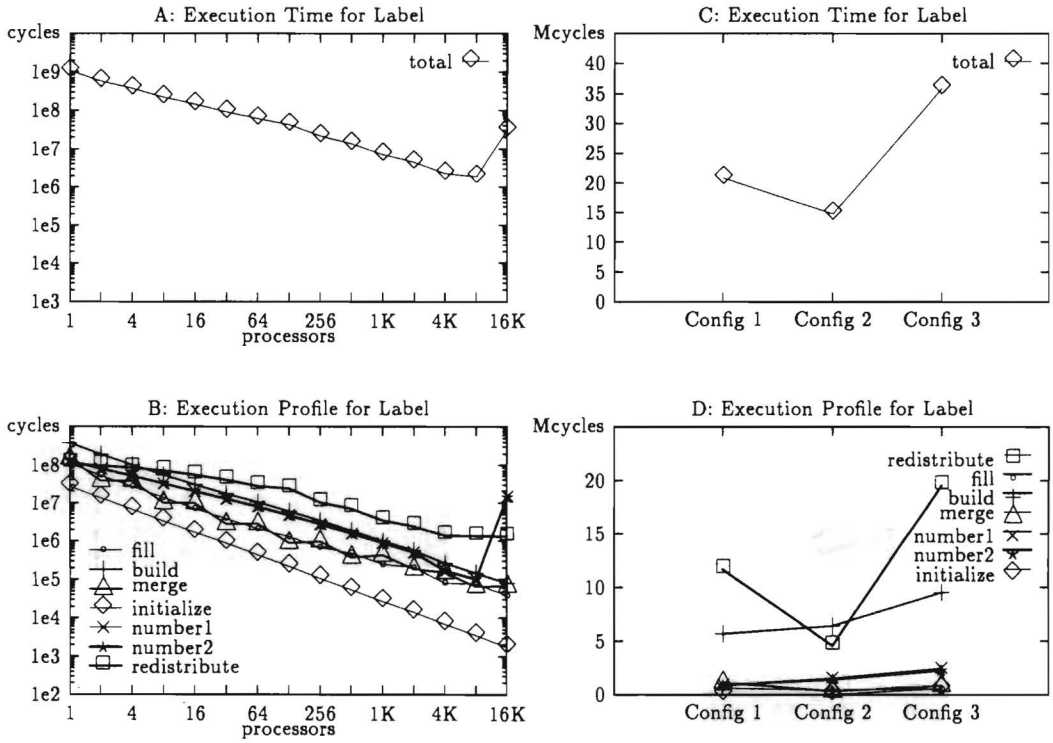

FIG. 5. Execution time and profile for the `label` program.

nitude speedup. Some of the phases show a step-like behavior. This effect is due to the use of an algorithm that scans the image horizontally. As the number of PEs is increased, the number of pixels in the image partition for each PE is reduced first in the X dimension, then in the Y dimension. The horizontal scans work in the X dimension, so these transitions provide significantly more speedup.

Figures 5C and 5D show the execution time and profile for the `label` program on the three static configurations. These figures show that configuration 2 performs the best for this program. This result occurs because the redistribute phase dominates the execution of this program. This phase is potentially erratic because its behavior depends on the distribution of border pixels among the PEs and it is communication bound. For a large number of PEs, the potential for a large deviation in the number of border pixels per PE can cause an increase in the amount of communication necessary. The other phases are split between those that perform best in configuration 2 and those that perform best in configuration 1. The build phase is similar to the `filter` program in its structure, and thus exhibits a similar characteristic. This program could benefit from processor reconfiguration, but as seen in the graphs, the benefits would not be dramatic.

Figures 6A and 6B show the execution time and profile for the corners program for 1 to 16K Config 3 PEs. These results are a little surprising because the corners program has an order of magnitude less data than the `filter` program, and potentially erratic communication patterns. These results do make sense, though, when one considers that this program still maintains at least 2 data points per PE in the 16K PE case, and these data points have been conveniently distributed by the `label` program. As for the communication, this program requires much less than the `label` program because each point on a region border need only communicate with those pixels a small distance along the border, and in many cases these pixels are on the same PE, or a nearby one. Figures 6C and 6D show the execution time and profile for the `corners` program on the three static configurations. This program favors configuration 1.

Finally, Figures 7A and 7B show the execution time and profile for the `rectangles` program for 1 to 16K Config 3 PEs. Again, these results are rather surprising, as this program uses only about 350 data points! Like the `label` program, this program achieves much less speedup than the fine grain `filter` or `corners` programs: only about two orders of magnitude for 16K PEs. The curves are a little erratic, and there is a definite reduction in slop after about 128 PEs—probably due to the smaller number of data points. This program does continue to improve its performance all the way up to 16K PEs. The explanation is that the data for this program is extremely unbalanced. It turns out the corners found in an image are necessarily very close in the image, thus the probability is high that many corners fall on a single PE, while other PEs have no corner points at all. Thus, long before the number of PEs exceeds the number data points, many PEs are idle during most of the computa-

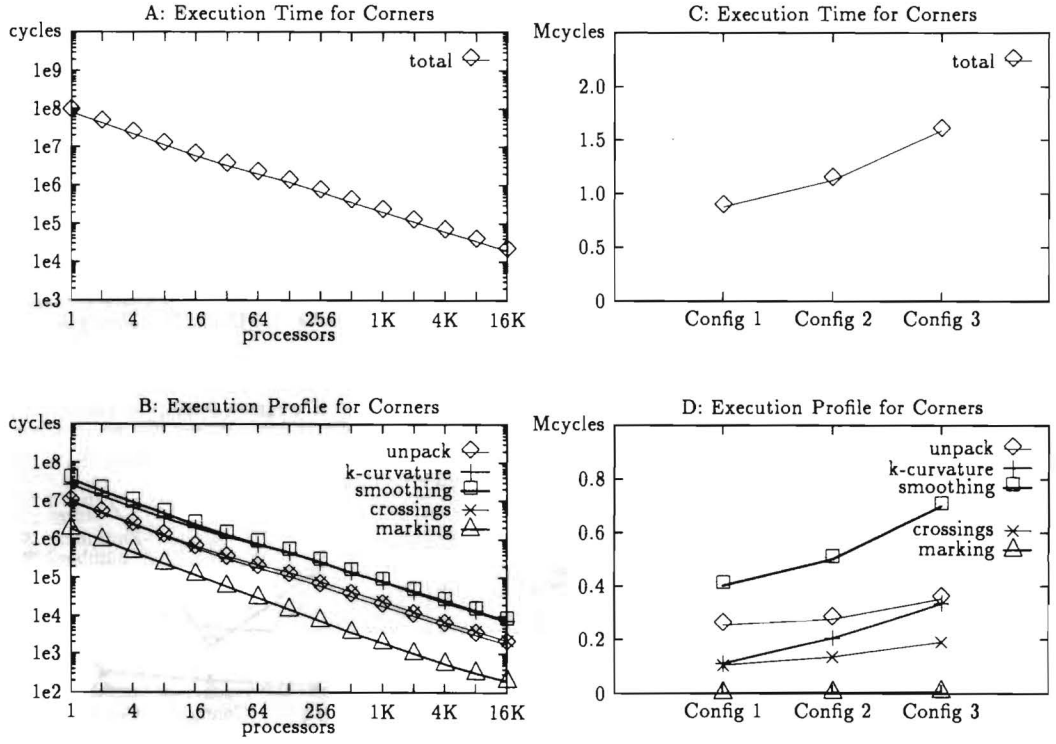

FIG. 6. Execution time and profile for the `corners` program.

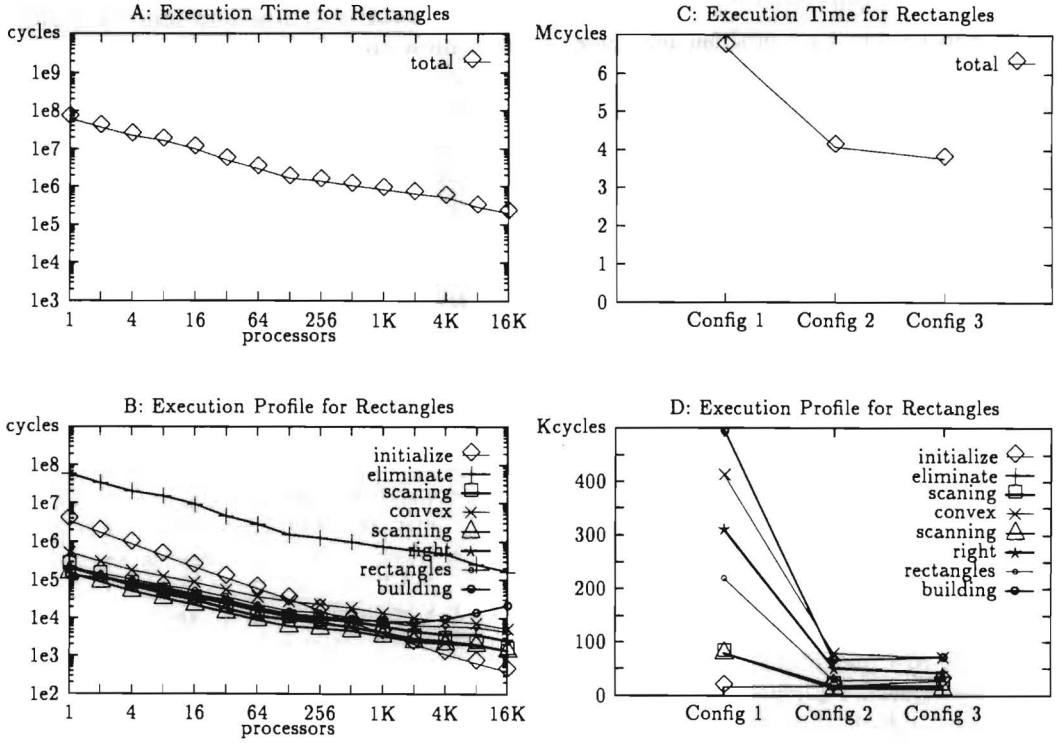

FIG. 7. Execution time and profile for the `rectangles` program.

tion. As the number of PEs increases, the number of data points per PE decreases, but at a considerably slower than the number of PEs in increasing. Each time more PEs are added, most of the new PEs are idle. This explains the extremely shallow slope of the graph. A redistribution algorithm might alter these curves, but the dominant phase of the program, the eliminate phase, is the one charged with eliminating the noncorner nodes from the boundary lists at the beginning of the program and thus uses as many data points as the `corners` program.

Figures 7C and 7D show the execution time and profile for the `rectangles` program on the three static configurations. The `rectangles` program favors configuration 3 by a wide margin over configuration 1. The profile shows this to be true for all phases save the initialization phase, though some of the other phases perform as well or even slightly better using configuration 2. This result is reasonable because clearly the program makes only modest use of additional processing elements, but the complexity of its computations can easily make effective use of more powerful processing elements.

Figure 8 shows the aggregate execution time of the four programs for each of our three homogeneous configurations. This figure shows that the effect of the `filter` and `label` program completely overshadow that of the `rectangles` program, and further that the `label` program's characteristic dominates that of the `filter` and `corners` program, to indicate that configuration 2 is the fastest of the configurations. Clearly, there is enough variation in the system that a multigauge architecture is worthy of consideration. Figure 8 also shows the performance of a multigauge reconfigurable architecture on the programs. There are two points plotted. The first shows the execution time for the four programs allowing reconfiguration between the four different programs as shown in Table III. This results in about a 20% improvement in performance. The second point shows the execution time of the four programs allowing reconfiguration between the phases of the programs as also shown in Table III. Here, a nearly 30% improvement is achieved.

3.3. Discussion

The results presented in this section provide a quantitative measure of the benefit of multigauge heteroge-

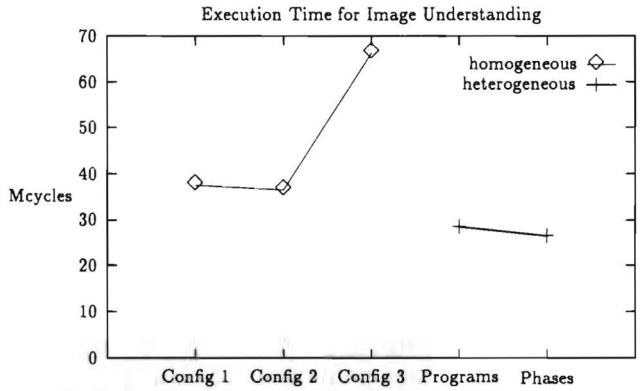

FIG. 8. Execution time of all programs on homogeneous versus heterogeneous architectures.

TABLE III
Configurations Used by Each Program and Phase

Program/Phase	Configuration	Program/Phase	Configuration
Filter	1	Corners	1
Communicate 1	1	Unpack	1
Median	1	*k*-Curvature	1
Fix Corners	1	Smoothing	1
Communicate 2	1	Zero Crossings	1
Sobel	1	Mark Corners	1
Label	2	Rectangles	3
Initialize	1	Initialize	1
Build Borders	1	Eliminate Nodes	1
Redistribute	2	Scan for Corners 1	3
Number Regions 1	1	Convex Hull	3
Merge Borders	2	Scan for Corners 2	3
Number Regions 2	1	Find Right Angles	3
Fill Regions	1	Find Rectangles	2
		Build Rectangle DB	2

neous capability for low and mid level computer vision applications. Reconfigurable architectures were studied in these experiments, which allow a maximum degree of flexibility, at the cost of custom hardware support. The fact that the same computational resources are used in each configuration means that interconnection issues are less significant in contributing to the cost of coordinating the various phases than would be the case if a nonreconfigurable heterogeneous system was used in these experiments. Extending this study to such statuc heterogeneous systems is part of our ongoing research. Finally, there are several other application domains such as computational fluid dynamics and optimization which could benefit from the flexibility offered by heterogeneous parallel systems. Our future research includes exploring these domains as in the context of both heterogeneous and reconfigurable architectures. The PCI model and RAW can be keys to understanding the program performance implications for such applications.

4. CONCLUSION

A clear understanding of the performance relationship of parallel programs and parallel architectures is essential to the successful implementation of heterogeneous parallel systems. We have taken one step in the development of this understanding by proposing the PCI model of parallel programs and parallel architectures which provides a basic means for reasoning about this relationship. Our initial work with this model has been to utilize it in experimenting with the performance implications of multigauge heterogeneous systems in the form of processor reconfigurable architectures. To do this, we have implemented RAW, a simulation environment based on the PCI model, which facilitates experimentation. Our experiments examine the use of processor reconfigurable architectures in low and mid level vision applications and show that such multigauge architectures provide better performance than any one homogeneous static architecture. Our ongoing research addresses other aspects of heterogeneous system design in the context of computer vision such as interconnections (both multiprocessor and networked multicomputers), different control regimes (such as SIMD, MIMD, and MSIMD), and multigranular systems composed of networked parallel systems of different granularities.

REFERENCES

1. Dally, W. Performance analysis of *k*-ary *n*-cube interconnection networks. *IEEE Trans. Comput.* **C39,** 6 (June 1990), 775–785.
2. Hillis, W. D. *The Connection Machine.* MIT Press, Cambridge, MA 1985.
3. Kartashev, S. I., and Kartashev, S. P. A multicomputer system with dynamic architecture. *IEEE Trans. Comput.* **C28,** 10 (Oct. 1979), 704–721.
4. Ligon, W. B. An empirical analysis of reconfigurable architectures. Doctoral thesis, Georgia Institute of Technology, Atlanta, 1992.
5. Ligon, W. B., III, and Ramachandran, U. An empirical methodology for exploring reconfigurable architectures. *J. Parallel distributed Comput.* **19,** 4 (Dec. 1993), 323–337.
6. Ligon, W. B., and Ramachandran, U. A reconfigurable supercomputer architecture. Tech. Rep. GIT-ICS-89/13, Georgia Institute of Technology, Feb. 1989.
7. Ligon, W. G., and Ramachandran, U. Simulating interconnection networks in RAW. *Proc. International Parallel Processing Symposium,* IEEE, New York, Apr. 1993, pp. 268–275.
8. Sejnowski, M. C., Upchurch, E. T., Kapur, R. N., Charlu, D. P. S., and Lipovski, G. J. An overview of the Texas reconfigurable array computer. *Proc. 1980 AFIPS National Comput. Conference.* AFIPS Press, Arlington, May 1980, pp. 631–641.
9. Siegel, H. J., Siegel, L. J., Kemmerer, F. C., Mueller, P. T., Jr., Smalley, H. E., Jr., and Smith, S. D. PASM: A partitionable SIMD/MIMD system for image processing and pattern recognition. *IEEE Trans. Comput.* **C30,** 12 (Dec. 1981), 934–947.
10. Synder, L. Introduction to the configurable, highly parallel computer. IEEE Computer, pages (Jan. 1982), 47–56.
11. Weems, C., Hanson, A., Riseman, E., and Rosenfeld, A. The DARPA image understanding benchmark for parallel computers. *J. Parallel Distributed Comput.* **11,** 1 (Jan. 1991), 1–24.
12. Weems, C. C., Levitan, S. P., Hanson, A. R., Riseman, E. M., Nash, J. G., and Shu, D. B. The image understanding architecture. *Internat. J. Computer Vision* **2,** 3 (1988), 251–282.

WALTER B. LIGON III received his Ph.D. in computer science from the Georgia Institute of Technology in 1992 and is currently an assistant professor of computer engineering at Clemson University. At Georgia Tech he was part of a team that developed system software for the SPOCK parallel processor and also developed the Reconfigurable Architecture Workbench. Currently he is the head of the Macintosh Telemetry and Control project at Clemson and is also conducting research in high performance I/O for parallel computer systems under a grant from NASA. His primary research interests are in computer architecture, parallel processing, and compiler design.

UMAKISHORE RAMACHANDRAN received his Ph.D. in computer science from the University of Wisconsin, Madison in 1986, and is currently an associate professor in the College of Computing at the Georgia Institute of Technology. He has researched extensively in the architectural design, programming, and analysis of parallel and distributed systems. He was a Co-Principal Investigator for the Clouds distributed operating system project at Georgia Tech. His current interests are in investigating software and hardware mechanisms for building scalable shared memory systems, and studying the scalability of parallel systems from an applications perspective. He is the recipient of a Presidential Young Investigator (PYI) Award from the National Science Foundation (NSF) in 1990, and the Georgia Tech doctoral thesis advisor award in 1993.

Received January 1993; revised October 27, 1993; accepted October 27, 1993

JOURNAL OF PARALLEL AND DISTRIBUTED COMPUTING 19, 323–337 (1993)

An Empirical Methodology for Exploring Reconfigurable Architectures

W. B. LIGON III AND U. RAMACHANDRAN

College of Computing, Georgia Institute of Technology, Atlanta, Georgia 30332-0280

Recent research in reduced instruction set computer architectures has emphasized the importance of the empirical approach to designing computer architectures: architectural features are analyzed for utility and cost with respect to the system software that uses them. This approach has resulted in architectural simulators that allow computer designers to vary the features of the architecture being simulated and to analyze how the addition or removal of these features affects the cost and performance of the architecture. In this paper we apply this technique to a new area: reconfigurable architectures. Our approach is to use an empirical methodology that emphasizes the interaction between the target software and the reconfigurability features of parallel architectures. We have developed a set of tools, the reconfigurable architecture workbench, that assists in this methodology by allowing parallel programs to be simulated on a target architecture in order to study the performance implications of various reconfigurability features. The workbench is based on a framework, the PCI model, which describes the range of parallel programs, parallel architectures, and reconfiguration features. We present details of the design and implementation of a prototype workbench, GT-RAW. GT-RAW is being used to study the utility of one dimension of reconfiguration for image processing and image understanding applications. We present an example of the experiments that are being conducted with GT-RAW as a demonstration of our empirical methodology. © 1993 Academic Press, Inc.

1. INTRODUCTION

Recent research in computer architecture has focused on parallel processing as a means for achieving high performance. This research has yielded a number of different designs which vary along three distinct dimensions: the number and size of the processors, the type of control strategy used, and the interconnection between the processors. For example, the Connection Machine [8] is a parallel processor with 64K one-bit processors, an SIMD control strategy, and both a mesh and a hypercube interconnection between its processors. The Cray X-MP computer [2], on the other hand, has four 64-bit processors with multiple floating-point pipelines, an MIMD control strategy, and a shared memory. Given the rich domain of parallel applications it is always possible to find a set of applications that perform well on a given parallel architecture. However, it is often difficult to determine which architecture is best for a given application. Furthermore, it is not clear that any one architecture is capable of achieving a high speedup over a range of applications. The problem is that different parallel architectures exploit different kinds of parallelism in order to achieve speedup. Unfortunately, not all applications provide the same amount or type of parallelism. For example, some applications may exhibit data-level parallelism, while others may exhibit task-level parallelism. Also, not all applications require the same communication patterns between parallel execution threads. These features can affect the ability of a given architecture to achieve its full potential.

Thus the architectural requirements of classes of applications differ. This fact has paved the way for recent research into reconfigurable architectures. Such architectures provide the capability to alter the number and size of each processor, the control strategy, and/or the processor interconnection. Examples of reconfigurable architectures are the CHiP architecture [24], image understanding architecture [27], reconfigurable meshes [18], and the polymorphous torus [13] which can alter their processor to processor connections; PASM [22] and TRAC [21] which can alter their control strategy from MIMD to SIMD and various hybrid strategies; and the dynamic computer group [11] which can combine many small processors into a few larger processors.

In theory, reconfigurable architectures overcome these problems by allowing the architecture to dynamically conform to the needs of a specific application. In practice, however, reconfigurable architectures allow only a few degrees of freedom due to the complexity involved in implementing such architectures. It is therefore essential to understand what features will provide the best performance increase over the target range of applications. Unfortunately, it is virtually impossible to determine the features of a reconfigurable architecture that resulted in a specific performance gain, since it is often difficult to isolate the effect of the architectural features on the performance. One feature may obscure the poor performance of another. For example, a histogram operation is not particularly well suited to a computer with thousands of one-bit processors. However, if such a computer is

0743-7315/93 $5.00

equipped with a hypercube routing network it may be able to efficiently implement the histogram.

Unfortunately, very little research has been done to quantify the performance implications of the different types of reconfigurability over a range of applications. Hence it is difficult to decide what kind of reconfigurable architecture is suitable for an application or how to choose between different reconfigurability features. The trend until now has been to benchmark a reconfigurable architecture and compare it against a non-reconfigurable one. Such a comparison reveals very little information as to how individual features affect performance and provides very little insight to the designers of new architectures in deciding the kinds and degrees of reconfigurability that would be effective for a range of applications. The problem is further compounded by the fact that there is little expertise in programming such machines, so it is not always clear that the features have been fully utilized.

Our approach is to apply an empirical methodology, similar to RISC research [19], to the design of reconfigurable architectures. We employ a methodology that centers around a set of experiments that match the execution behavior of a parallel program to the limitations of a given parallel architecture. These experiments are iteratively refined in order to determine a set of features that are useful for providing performance improvements to the programs studied. This methodology is outlined as follows:

1. A target group of applications is identified. The key to the methodology is that representative software be available for analysis.
2. Experiments are conducted to identify the general classes of architectures that hold promise for the target applications. As a first cut, the system designer may specify the most likely candidates based on some *a priori* knowledge about the target application. The candidate architectures may include commercially available ones.
3. The experimental results are analyzed to identify the best candidate architectures. In the event that more than one architecture proves to be well suited, but for different parts of the application space, some type of reconfiguration may be deemed appropriate.
4. More precise experiments are performed to determine the types of reconfiguration that may boost the application performance. Several iterations of this step and the previous one may be used to incrementally reduce the search space.
5. Steps 3 and 4 are eventually expected to lead to a target architecture specification that may be worth exploring further from the point of view of design feasibility.
6. More precise models may be realized to match the design of the target architecture, and the applications may be analyzed with respect to these models. This step would help uncover problems in the design, as well as evaluate cost versus performance of the reconfiguration features incorporated in the design.
7. The design may be refined to address the problems uncovered, incorporate custom hardware, or alter the parameters of reconfiguration. This step and the previous one are used iteratively to the extent appropriate.
8. The overall design may now be evaluated to determine the feasibility of developing a prototype.

It should be noted that while this methodology is similar to that used in RISC research, the level at which we are modeling parallel architectures is different. Our research is focused on the larger parallel structure of architectures, such as the processor granularity, the control strategy, and the interconnection network. We do not study the features of the processor instruction set, as is done in RISC research. Consequently, our work is intended to complement RISC research.

In order to realize this methodology, we need to provide a means for analyzing the performance of applications on a testbed that allows various architectural features to be added, or removed, or allowed to vary (i.e., to be reconfigurable). This testbed must be able not only to give a simple performance measure, but also to identify what parts of a program are well suited to the architecture and what parts are not; and what parts of an architecture are well suited to the application and what parts are not. This goal is achievable by obtaining a performance profile that indicates execution time and speedup and a utilization profile that indicates the utilization of processors, control units, communication links, and the various configuration modes available.

2. THE RAW ARCHITECTURE

To this end, our research is directed towards the design of the reconfigurable architecture workbench (RAW) (Fig. 1), a simulation platform for investigating the relative utility of architectural reconfigurability with respect to a class of applications. We have developed a prototype workbench, GT-RAW, which we are using as part of our research. The workbench is a set of software tools that allows the user to analyze the match between architectures and programs. The toolset includes a source code parser for converting a program into an interpretable architectural intermediate code (AIC); a model of the target architecture, the architectural descriptive code (ADC); and a simulation engine. The simulation engine analyzes the dynamic behavior of the program on the target architecture by executing the AIC with an interpreter and analyzing the execution on-the-fly with a trace analyzer. This design is unique in that the trace analyzer feeds back timing information derived from the ADC to guide the

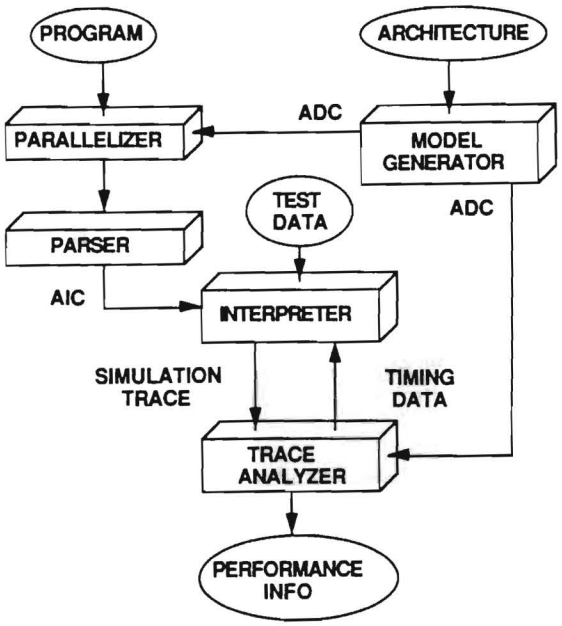

FIG. 1. RAW architecture.

program's execution in an architecture dependent manner.

The heart of the workbench is the representations of the program and the architecture: the AIC and the ADC, respectively. In essence, these two components work together to model the interaction between software and hardware. The AIC is an architecture-independent representation of a parallel program based on a simple interpreted machine model. The ADC is a description of the resources available in the parallel architecture and serves to govern the execution of the AIC representation of the program. The AIC and ADC are based on a machine model (to be described next) that serves to describe the classes of programs and architectures that can be studied with RAW.

2.1. The PCI Model

In order to develop the components of GT-RAW, we need to have a framework for understanding how to represent our programs and architectures in such a way that they can be properly simulated by our workbench. This framework should define the features of programs that are simulated, the features of architectures that are modeled, and the different classes of reconfigurability that are studied. These definitions describe the structures that must be provided by the various tools of the workbench in order to support the modeling of these features. These three definitions also outline three classification schemes for programs, architectures, and reconfiguration that allow us to characterize the requirements of parallel programs, the features of parallel architectures, and the capabilities of reconfigurable architectures. These schemes combine elements of noted classification schemes such as Feng [5], Handler [7], Flynn [6], Snyder [25], Enslow [4], and Kung [12]. We call this framework the processor, control, interconnect (PCI) model.

The PCI model defines parallel programs as being composed of three components: (1) independent *instruction streams* each of which execute in a sequential manner; (2) independent *data streams*, each of which execute instructions provided by an instruction stream on different data elements; and (3) *communication* between the data streams and synchronization between the instruction streams which may take place either by message passing or via shared memory. PCI defines parallel architectures as being composed of three similar components: (1) *control units* that process instruction streams, (2) *processing elements* (PE) that execute the instructions issued by a given control unit, and (3) *interconnection links* that allow synchronization among control units and exchange of data among PEs (and their memories). We call these components the control configuration, the processor configuration, and the interconnection configuration. Finally, PCI defines three classes of reconfigurability: (1) processor reconfiguration allows the architecture to trade off the number of PEs for the speed of the PEs, (2) control reconfiguration allows the assignment of PEs to control units to be changed, and (3) interconnection reconfiguration allows the communication topology of the parallel architecture to be modified.

There are four components of GT-RAW that work together to produce experimental results. Two of these are the representations of program and architecture (AIC and ADC), and the other two are the active components responsible for producing the architecture dependent execution (interpreter and trace analyzer). Each of these components contain structures based on some part of the PCI model. The act of executing a program with the GT-RAW simulator combines these structures in order to produce the desired results. The next four sections describe these four components in detail. Finally, we describe how these components work together during execution.

2.2. The AIC

A program encoded in AIC consists of segments of a sequential stack-machine assembly code embedded with calls to primitives that define the parallel structure of the program. The major primitives are as follows:

- The `fork` primitive creates a new instruction stream at a given point in a program and makes a copy of the data for that stream. Completed instruction streams may `exit` and other instruction streams may `wait` for that event.

• Data streams are allocated with the `pealloc` primitive which causes SIMD style processing to begin. These data streams can be selectively activated with the `pemask` primitive. Each data stream independently evaluates the argument to this primitive and only those for which a true value is produced continue to process instructions. `peflip` causes all data streams deactivated by the last `pemask` to be activated, and vice versa. `peunmask` causes all data streams deactivated by the last `pemask` or `peflip` to be reactivated. Masking context is maintained on an implicit stack.

• Messages are processed with `send` and `receive` primitives. The `send` primitive is nonblocking and allows an arbitrary amount of data to be transmitted to a specified destination. The `receive` primitive allows messages to be received from either a specific source, or any source. The `receive` primitive can be made to block indefinitely, timeout after a specified time period, or return immediately in the event a message is not available.

• GT-RAW's memory organization provides a global shared memory, a memory that is shared between data streams assigned to a given instruction stream, and a memory that is private to each data stream. Stack data is in private memory. Static data is available to all data streams of a given instruction stream. The GT-RAW parser also provides a "poly" data class that allows static data to be placed in each data stream's private memory. An instruction stream's memory consists of 16 segments of virtual memory, a few of which are automatically allocated (such as code segment and static data segment). A `memalloc` primitive allows a program to allocate additional segments. A variation of the `fork` primitive, `sfork`, allows the program to specify segments that are to be shared between instruction streams. The `getsegaddr` and `setsegaddr` primitives can also be used to set up shared segments.

Additional primitives are also provided to allow a program to interact with the architectural aspects of the simulation. The effects of these primitives are to alter aspects of the architecture dependent execution of the program, but not necessarily making the program representation architecture dependent.

• Another variation of the `fork` primitive, `pfork`, is used to specify instruction streams that should be resident on the same control unit. This allows the specification of concurrent, rather than parallel execution. Concurrent instruction streams are subject to the detail of the control unit scheduling policy and generally result in a different execution.

• The `memdist` primitive specifies that data objects should be distributed over a number of memory modules. Memory modules are the terminal points in the PE to memory interconnection network of a shared memory system and are therefore part of the architecture, not the parallel program (in contrast to the memory segments, which are part of the program). In RAW, these modules are used to simulate traffic on the interconnection network and do not actually contain any data items. The `memdist` primitive specifies how the elements of a data structure are allocated among the memory modules and thus affects the traffic patterns caused by accessing the objects.

• Finally, the `setarch` primitive is provided to allow the running program to alter the active ADC. This does not affect the execution of the program but will affect its performance.

Additional functionality such as special scheduling algorithms and hardware supported synchronization primitives can be specified in software on top of the given primitives.

2.3. The Interpreter

The interpreter is basically an event handler and a simulator for the AIC's stack machine instructions. The interpreter maintains data structures that represent the components of an executing AIC program: instruction stream control blocks, data stream control blocks, message control blocks, and memory segments. These structures are created, destroyed, and manipulated by the event handler and AIC instruction simulator. The events processed by the interpreter are generated by architecture components of the trace analyzer (described next) and cause the execution the AIC representation of the program. Three primary events characterize the execution of the parallel program represented by the AIC.

• the `instruction_issue` event, generated by the control units, causes the instruction streams to be processed,

• the `instruction_execution` event, generated by the PEs, causes the instruction simulator to perform the given instructions, and

• the `message_delivery` event, generated by the communication links, moves message structures in and out of the instruction and data stream control blocks. The `message_delivery` events are also used whenever a memory instruction attempts to access memory modules via an interconnection network.

The instruction simulator does not functionally simulate the target architecture, but rather implements the stack machine of the AIC. Recall that our research does not attempt to study architectures at a level of detail that includes the effects of the processor instruction-set. As such, the interpreter simply executes the AIC's instruction set and this is considered to be equivalent (in time) to any other instruction set. On the other hand, our research

does consider architectural issues that would affect the overall performance of the parallel program. These issues include the processing power of the PEs relative to the number of PEs; the word width; and the availability of specialized hardware such as floating point units. For example, bit-serial architectures may require several thousand instructions to be executed to perform a single floating-point operation in each PE. A bit-parallel architecture with floating-point hardware, on the other hand, may execute the same operations in a few cycles. The architecture independent execution of these operations use the features of the underlying system that hosts the workbench, but as we shall see, the effects on performance analysis and instruction timing may differ for the two architectures. The timing analysis would take into account the level of parallelism that exists in the architecture. For example, while an individual bit-serial operation may be expensive, a number of these may be executed in parallel. By the same token, while an individual bit-parallel operation may be quicker, a comparable architecture (with the same amount of hardware resources) may not have the same level of parallelism as the bit-serial one.

Although the basic execution of instructions is not affected by the target architecture, the overall execution of a parallel program on two different parallel architectures may differ in the way instruction streams, data streams, and communication are interleaved. The workbench interpreter simulates the simultaneous occurrence of an arbitrary number of instruction issues, instruction executions, and message deliveries. The way these events are interleaved is controlled by the timing information computed by the trace analyzer using the ADC. Any number of these events that are scheduled to occur at the same time are executed sequentially (since RAW runs on a uniprocessor). The order of execution of these parallel events is non-deterministic. A correctly parallelized program should be insensitive to this order of execution.

2.4. The Trace Analyzer

The trace analyzer is a collection of functions that work together to schedule events for the interpreter producing an architecture dependent execution of an AIC program. In the absence of a specific ADC, the trace analyzer simulates a machine that provides an infinite number of control units, PEs, and communication links. An ADC model of an architecture (described next) places limits on the resources available to the AIC program by specifying the number of control units, PEs, and interconnection links available. As an example, an ADC model of a hypercube MIMD machine might define a fixed number of N control units, associate one PE with each control unit, and provide links of a given bandwidth between the appropriate nodes. A given program coded in AIC may choose to define a single instruction stream, and allocate N data streams to that process, but on the MIMD architecture, the N data streams would be executed sequentially. Similarly, a hypercube SIMD machine might have a single control unit and N PEs available to that control unit. If a given program attempted to allocate N processes, each would have to execute sequentially on the architecture. Similar examples can be given for the interconnection architecture, where nodes that attempt to communicate but do not have a link defined between them must pass through other nodes and incur extra delays.

The trace analyzer maintains structures that model the control units, PEs, and communication links of the architecture, much like the interpreter does for instruction streams, data streams, and messages. In order for events to be scheduled, the affected program structure must be assigned to an appropriate architecture structure. As in the above example, there may exist more instruction streams than control units, but in order for an instruction issue event to be scheduled, the instruction stream must be assigned to a control unit.

The trace analyzer is invoked when the executing AIC program calls a primitive that creates a new control block (instruction, data, or message). Functions provided by the trace analyzer manage the allocation of the architectural resources to the program components in accordance with the specifications contained in the ADC. Once a program component has had an architectural component allocated to it, the events that cause the execution of the program are scheduled. Timing information for these events is obtained from lookup tables in the ADC via additional functions provided by the trace analyzer. These functions compute the time between successive instruction issues, the time until an instruction has completed execution, and the time to traverse a communication link. These timings are accumulated by the trace analyzer to produce performance and utilization profiles while the interpreter maintains the global execution time.

2.5. The ADC

The ADC is a data structure that represents the various architectural configurations that can be modeled by the workbench. These data structures are used by the trace analyzer to produce timing information for the interpreter and to perform resource management. In particular, the ADC provides the architectural information required to generate timing characteristics for the primitives in the AIC. The ADC models each of the three dimensions of the PCI notation independently:

- The *processor configuration* is modeled by specifying a description for each PE type that can be configured

in the underlying architecture. A description of a given PE type includes a timing chart with an entry for each instruction in the AIC and the amount of hardware resources required to implement an instance of that PE type. Each entry in the timing chart includes two values, the *latency* and the *frequency* which respectively indicate the execution time and the level of pipelining for the corresponding AIC instruction. For example, a machine might consist of a number of one-bit hardware resources. A PE type might be a one-bit PE (such as used in the Connection Machine) requiring a single resource. The timing chart entry for an integer add instruction might have a latency of four times the number of bits in the operand (two fetches, one compute, and one store per bit) and an identical frequency (which indicates no pipelining). Another PE type might be a 32-bit PE requiring 32 resources. This PE's timing chart entry for the same integer add instruction would have a latency of five (an extra cycle to compute the carry) times the number of bits in the operand divided by 32. A processor configuration consists of all the distinct PE descriptions and the numbers of each type of PE available in the architecture while in that configuration. The sum of all of the resources used in any given configuration must equal the total amount of the given architecture.

Special features can be modeled with a "warp mode" that allows arbitrary functions to consume a finite amount of time. For example, if a PE has a hardware *sine* functions that operates in 5 cycles, a *sine* function can be implemented in software and the `pewarp` primitive can be used to "turn off the clock" so that its execution requires only 5 cycles. Processor reconfiguration is achieved by re-arranging the distinct types of PEs, subject to the total hardware resources in the processor architecture.

- The *control configuration* is modeled by specifying the number of control units in the system and a mapping of PEs to each control unit. Control units maintain a list of instruction streams and a list of PEs allocated to them. There is a need for a scheduling policy when there are more concurrent instruction streams active than control units to process them. The built-in scheduling policy is a non-preemptive one that allows an instruction stream to utilize a control unit until the process terminates or blocks, but users can choose to specify a new policy. Control reconfiguration is achieved by rearranging the allocation of PEs to control units.
- The *interconnection configuration* is modeled by a set of channel class descriptions and a network of channels. Each class description includes attributes that describe the performance of a channel: (1) the bandwidth of the channel in bits, (2) the latency of the channel, and (3) the setup time of the channel (which includes tear-down of any existing connection). The channel network is a matrix of channel descriptors that include a reference of the class of the channel, the source and sink of the channel, and a queue of messages waiting to utilize the channel. Messages traversing the network cause *message delivery* events to occur as each link processes the message. Messages are automatically passed to the next link in a multi-hop situation until the destination is reached. Connection reconfiguration is modeled by allowing the instances of the defined classes to be varied dynamically. Channels may also be set up to cause interrupt processing to occur at the sink, thus allowing inter-processor interrupts to be modeled. The AIC provides instructions for coding interrupt handling routines.
- *Reconfiguration costs* are modeled by the ADC as well. Each type of reconfiguration (processor, control, interconnection) may incur a cost set by the ADC as a result of executing the reconfiguration function. Processor reconfiguration may require data in the PE's memories to be re-formatted for use in the new configuration. The trace analyzer provides a function accessible to the user for specifying data that needs to be re-formatted with the cost specified by the ADC. Control reconfiguration requires that control units giving up PEs to another synchronize (i.e., complete their task) before the new control unit can begin processing. This cost is inherent in the PE management functions. Similarly, interconnection reconfiguration must wait for all affected links to finish processing all existing messages before the links can be redirected. These costs can be managed by the trace analyzer and are specified in the ADC.

2.6. *Execution on the Workbench*

Now that we have discussed each of the four major components of RAW, we can consider how these components work together to produce an execution of an AIC program representation on the workbench. When RAW begins, initialization routines call an interpreter function `create_instruction_stream` which allocates an instruction stream control block and a single PE control block. Next, a reference to the instruction stream is passed to a trace analyzer function `schedule_instruction_stream`. This function compares the number of existing control units to the maximum number allowed by the ADC, and if the maximum has not been reached (as should be the case) it creates a control unit structure and schedules the instruction stream on it. This routine calls another trace analyzer function `schedule_data_stream` which allocates a PE in a similar manner. Since the instruction stream is the only one allocated to this new control unit, an `instruction_issue` event is scheduled for the control unit, and execution begins.

When the interpreter determines that the `instruc-`

tion_issue event is ready to occur, it consults the trace analyzer via the function next_instruction to determine when the next instruction_issue event and instruction_execute event are to occur and schedules them, based on the timings provided. The trace analyzer consults the latency and frequency attributes in time tables of the ADC and must also compare the number of data streams allocated to the instruction stream versus the number of PEs available to execute the instructions in order to compute these values. When the instruction_execute event occurs, the interpreter invokes the instruction simulator to carry out the instruction.

The various primitives described in Section 2.2 are available to the AIC program through a monitor instruction. When a fork (or one of its variations) is executed, the function create_instruction_stream is called, and the process repeats itself. In the event a new control unit cannot be created, a new instruction stream will be allocated to a control unit, but must wait until the instruction stream using the control unit blocks or exits before an instruction_issue event of the new instruction stream can be scheduled. Similarly, the pealloc primitive translates into a create_data_stream which, in turn, calls schedule_data_stream. In the event a PE is not available, the instruction stream is not blocked; rather, the trace analyzer notes that PEs must do double duty, and thus it adjusts the event timings accordingly. Alternatively, the instruction stream may query its current resources and choose to block until it has acquired the desired resources.

Finally, when a send primitive is called, a message control block is created and passed to a trace analyzer function send_on_link. This function looks in a matrix indexed by the current location of the message and its ultimate destination and retrieves a reference to a communication link. If the communication link is available, then the trace analyzer determines the transit time on the link and schedules a message_arrival event accordingly. If not, the message is placed on a queue until the link is available, at which time the event is scheduled. When the message_arrival event occurs, the interpreter consults the link to establish a new current location, and the process repeats. When the the message's current location equals its ultimate destination, it is placed on the input queue of the appropriate data stream, where a call to the receive primitive can retrieve it. This same process is used to model accessing a remote memory via a network. In this scenario, when the message (an address) reaches its destination (a memory module) a special memory_fetch event is used to return the message (now a data value) to its source.

The RAW maintains usage and performance data for all components of the architectures and programs used in an experiment. This data can be used to show the execution time of all or selected parts of a program, the amount of time spent using different configurations, and the utilization or contention for resources. These measures can be shown as summary information, as a profile for different parts of the program, or as a histogram over the execution of a program. Eventually this data should be available in suitable graphical form, but for now is simply saved for later processing.

3. IMPLEMENTATION OF THE WORKBENCH

We are currently working with a prototype implementation of RAW called GT-RAW. A C parser is provided to compile the source code into AIC. The parallel and reconfiguration primitives of RAW as well as I/O facilities are provided via a monitor system implemented as part of the GT-RAW interpreter. One extension to the C language, a "poly" data class, has been provided to control the placement of data items in memory (see Section 2.2). This construct is implemented via a source code pre-processor. ADC code is specified by initializing a set of pre-defined data structures using standard C syntax. This code is compiled and loaded at run time along with the binary program code. The type definition mechanism for the C compiler provides the ability to define instruction and channel class descriptors. The event-processing loop of the interpreter is implemented using the *SMPL* event-driven simulation package [17]; GT-RAW itself is implemented in C and has been compiled to run on a number of Sun and Vax workstations.

GT-RAW is capable of handling all three aspects of parallel programs, all three aspects of parallel architectures, and all three types of reconfiguration specified by the PCI model. While all of these features have been implemented, some of them are still being tested and refined. All performance and usage data described above is gathered by the system. Listings of the simulation time and program profile are currently produced.

4. AN EXAMPLE OF THE EMPIRICAL APPROACH

Currently, we are pursuing the empirical methodology to investigate the utility of reconfigurable architectures. The methodology presented in Section 1 is straightforward. However, in practice one must consider a wide range of programs and architectures before any general conclusions can be drawn regarding the suitability of different architectures and reconfigurability features. Thus the methodology has to be applied over a space which represents the cross product of the target programs and architectures. Each point in this space involves conducting several experiments consisting of many program executions that result in a large amount of performance data.

Definitive conclusions can be drawn only after analyzing all of this information.

Our first step in understanding this methodology is to apply it to a small domain. This helps in reducing the space of architectures and target programs, thus making the validation of the methodology tractable. With this in mind, we have chosen to concentrate our efforts on image understanding systems. Similarly, we have decided to limit our initial investigation to the study of the effects of adding various reconfiguration features to an existing architecture. We believe this is a prudent approach for two reasons: it provides a solid starting point so that the ADC models can be verified by executing target programs on actual hardware, and it is more practical to estimate the cost of design changes to an existing machine than to estimate the cost of a totally new machine.

By choosing the DARPA Image Understanding Benchmark to provide a target set of programs [26] and using the analysis of the performance of the benchmark on various machines in [3], we have completed steps 1 and 2 of the methodology. Based on this analysis, we have selected a fine-grain SIMD architecture such as the Connection Machine as our base architecture. We are investigating the effects of adding processor, control, and interconnection reconfiguration to this base architecture. Currently, we have ADC models of the base architecture and its derivatives with various reconfigurability features added. We are also in the process of developing various versions of the benchmark programs for GT-RAW. As the experimentation progresses, we plan to further refine the ADC models for those architectures that show promise for these benchmark programs.

The scope of this paper is the description of our methodology and the tools we have developed to assist in carrying it out. For this reason, we have decided to present the analysis of a single reconfigurability feature for a specific program as a means of illustrating the nature of the experiments used in steps 3 and 4 of the methodology. While the results of these experiments are clear, the reader should note that they represent one data point among the many that must be evaluated before definitive conclusions can be drawn. The following is a presentation of these example experiments.

4.1. The Histogram Equalization Program

We have studied several programs all of which are used in image processing and image understanding systems. Three of these programs, mean filter, 2D discrete Fourier transform (DFT), and histogram equalization, have been studied previously in the context of reconfigurable architectures in [1, 10, 16, 23]. Other programs we are studying are those from the Image Understanding Benchmark such as region labeling, k-curvature, convex hull, and Hough transform. Our future plans also include studying programs from middle and upper level vision functions such as graph matching and top-down hypothesis verification. For the purposes of this paper we have decided to present the histogram equalization program as an example of how GT-RAW can be used to analyze the performance potential of processor reconfiguration. The histogram equalization program is interesting because it involves several phases which exhibit different algorithmic characteristics relative to the processor granularity. The phases of our histogram equalization implementation are based on well-known algorithms such as those presented in [9]. We are not aware of any previous work that describes a combination of these algorithms that performs a parallel histogram equilization and hence we present our algorithm in some detail.

A histogram is a collection of n buckets numbered $0, \ldots, n$, where n is the number of distinct pixel values possible in the image. The number of elements in each bucket i is the number of pixels in the image whose pixel value is i. The task of this program is to compute a global histogram of the pixel values of an input image and then to compute a new image whose histogram has been equalized. This is to say that the pixel values of the image have been adjusted to utilize the possible range of values more fully. This procedure may be used to enhance detail in an image by increasing the contrast between different shades of gray. The program as we have written it is composed of seven phases. Initially, we assume that the input image is partitioned evenly across p available processors.

1. In the first phase, each processor computes a partial histogram of the image partition it contains.

2. In the second phase, the goal is to reduce the number of partial histograms from one per processor to no more than the number of buckets in the histogram. This is accomplished in $\log_2(\lceil p/n \rceil)$ steps. In each step the active processors are divided into pairs and one processor in each pair sends its histogram to the other, which adds the incoming values to its own. The sending processors then become inactive. The remaining active processors repeat this step until there are $a \leq n$ processors active ($a = \min(p, n)$). These a processors will be the only active ones until phase 6. Figure 2 illustrates the process of reducing $p = 32$ partial histograms to $n = 8$ partial histograms. This is accomplished in two steps. Initially processors P0 to P31 are active. Step 1 (thin lines) reduces the number of partial histograms from 32 to 16 leaving processors P0 to P15 active. Step 2 (thick lines) reduces the number of partial histograms from 16 to 8 leaving processors P0 to P7 active.

3. In the third phase, the goal is to combine the $\leq n$ partial histograms into a single global histogram, with no

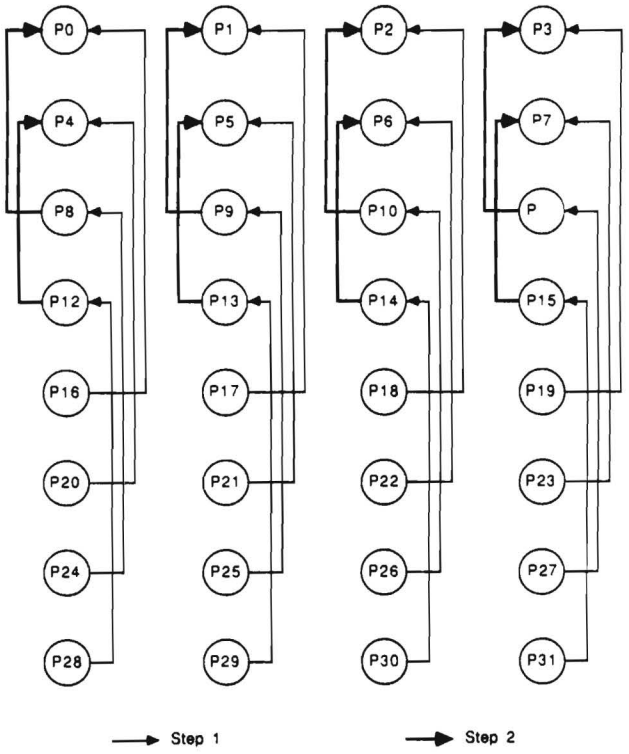

FIG. 2. Histogram equalization phase 2.

more than n/a buckets on each of a active processors. This is achieved in $\log_2(a)$ steps. Initially all of the active processors are in a single partition. In each step the processors in a partition are divided into pairs and the first processor of each pair sends half of its histogram buckets to the second processor, while the second processor sends the other half of the buckets to the first processor. The processors are then partitioned into those with the lower half of the histogram and those with the upper half. This process is applied recursively until each processor is in a partition by itself, and has $\leq n/a$ histogram buckets. Figure 3 illustrates this process for $n = 8$ histogram buckets and $a = 8$ active processors. The figure shows which buckets the processors have at the beginning of a step and which buckets they send to their partner in that step. In the example, at the end of the three steps, each processor has exactly one bucket of the global histogram.

4. In the fourth phase, the active processors compute a modified histogram where each bucket b is equal to the sum of all the histogram buckets, 0, ..., b. This is performed using the algorithm described in [9]. In each iteration i ($i = 0, \ldots, \log_2(a)$), each processor j sends its value to processor $j + 2^i$ and receives a value from processor $j - 2^i$, which is then added to its own value.

5. The fifth phase is the inverse of the third phase. The modified histogram is broadcast to the active processors using the reverse of the algorithm used in the third phase. The active processors are partitioned into pairs. The processors in each pair send their partner the buckets they are currently holding and receive the buckets from their partner, thus doubling the number of buckets on each processor. The partitions are then paired up and merged into half as many partitions. The processors in each of the new partitions are then divided into pairs such that one processor from each of the old partitions is present in each of the new pairs. This process continues until all of the active processors are in a single partition and each has a complete copy of the modified histogram. Figure 4 illustrates this process for eight histogram buckets and eight processors. Initially, it is assumed that each processor has exactly one bucket. The figure shows which buckets the processors have at the beginning of a step, and which buckets they send to their partner.

6. The sixth phase is the inverse of the second phase where the complete modified histogram is broadcast to all processors in the system. This is accomplished by having each of the active processors send its histogram to one of the inactive processors, which is then made active. This process repeats until all processors have the complete modified histogram. Figure 5 illustrates broadcasting

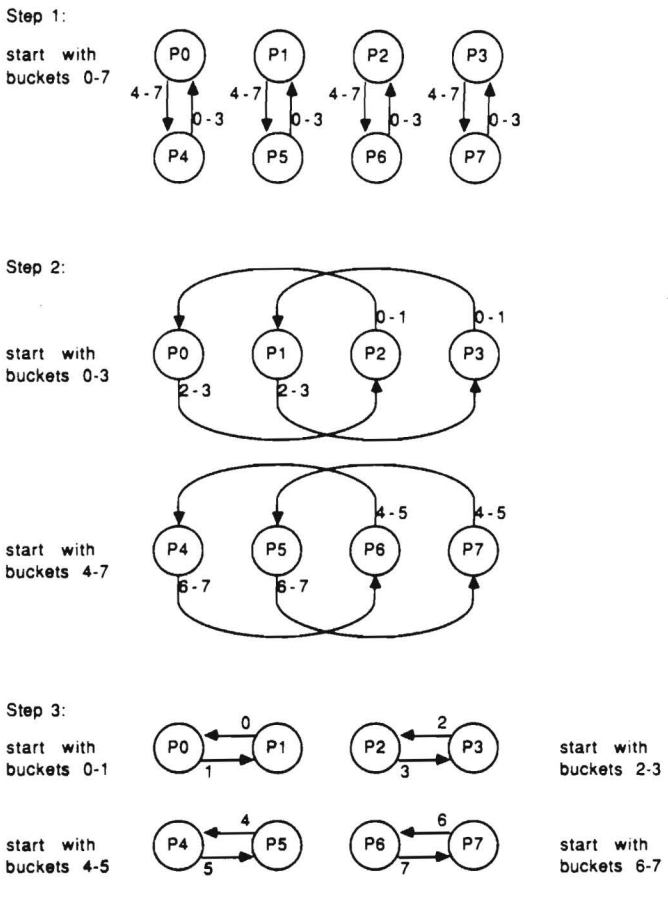

FIG. 3. Histogram equalization phase 3.

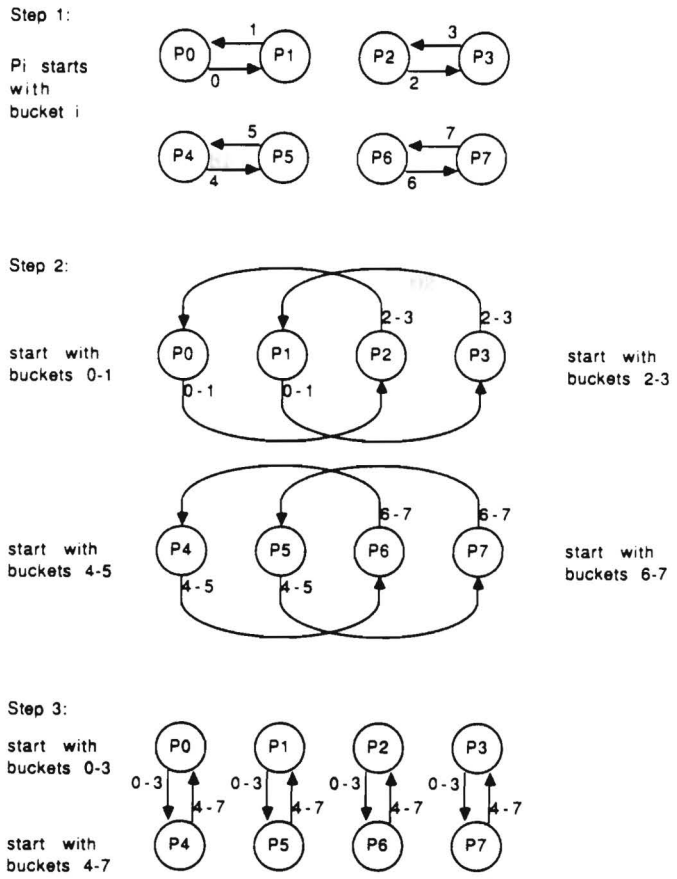

FIG. 4. Histogram equalization phase 5.

from 8 to 32 processors. This is accomplished in two steps. Initially processors P0 to P7 are active. Step 1 (thin lines) increases the number of histograms from 8 to 16 leaving processors P0 to P15 active. Step 2 (thick lines) increases the number of histograms from 16 to 32 leaving processors P0 to P31 active.

7. The seventh phase is the generation of the new image. Each processor computes a new value for each pixel in its image partition based on the old pixel value and the modified histogram.

This program is interesting to study with respect to processor reconfiguration because phases one and seven are distinctly different from phases three, four, and five. Phases one and seven have a large amount of parallelism, on the order of the size of the image but phases three, four, and five utilize only as many processors as there are buckets in the histogram. There is no interprocessor communication in phases one and seven, all data is local to each processor, whereas in phases three, four, and five, the tasks involve global computations and are primarily communication bound. Phases two and six serve primarily to interface between these two distinct modes.

4.2. Architectures

For the experiments presented in this section, all of the architectures studied are SIMD machines with homogeneous PEs and a hypercube interconnection network. The word size and capabilities of the PEs differ from one architecture to the next. The details of the different PE architectures are presented as the discussion progresses. However, it is worth noting here the origin and selection of the ADC models. The model used in the first experiment has no correlation to any existing machine. It is intended to analyze the available parallelism in the program by assuming the computation and communication times to be equal. The models used in the second and third experiments are based on reconfiguring a set number of Connection Machine PEs into larger logical PEs by the addition of some specialized hardware. The details of the reconfiguration mechanism are presented in [15] and a precise development of the models used can be found in [14]. A number of configurations can be developed using this mechanism. The configurations presented in Experiment 2 are chosen based on the results of Experiment 1. These architectural assumptions should be kept in mind when reading the following section.

4.3. Experiments

A series of experiments was conducted to analyze the behavior of the histogram equalization program. In each

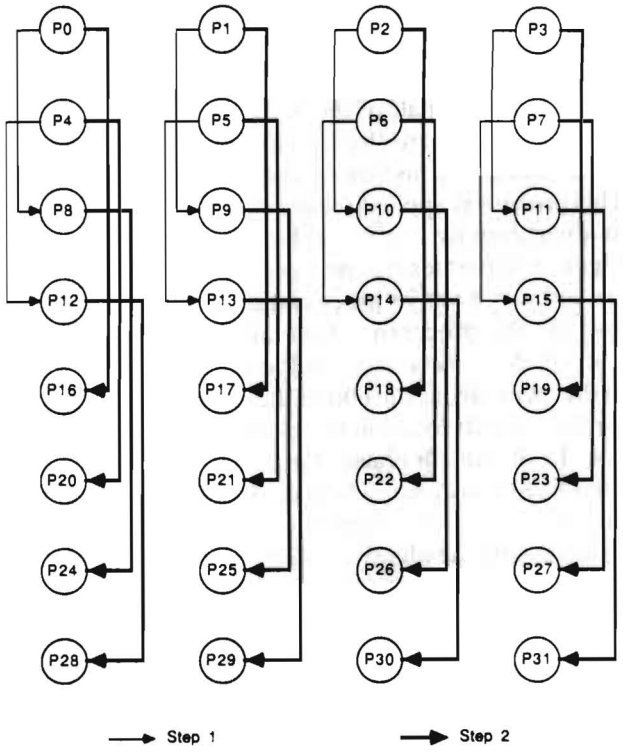

FIG. 5. Histogram equalization phase 6.

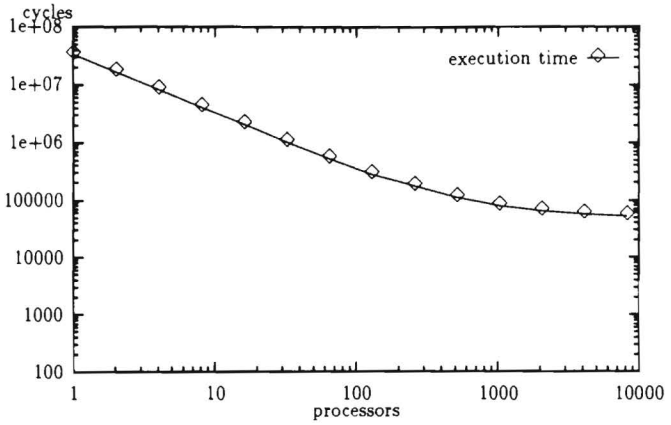

FIG. 6. Execution time for an increasing number of processors.

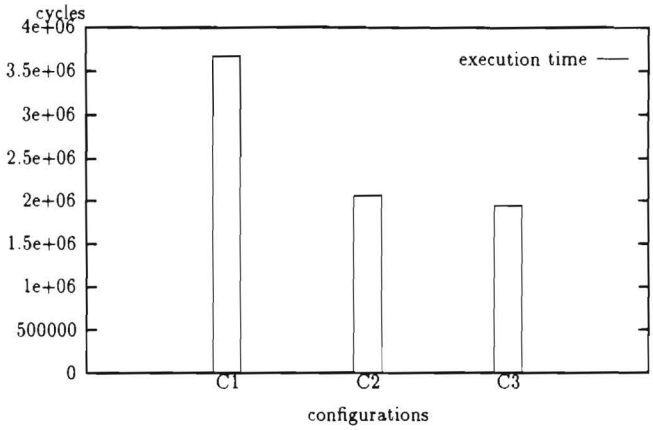

FIG. 8. Execution time for three processor configurations.

experiment, the image size was 512 × 512 pixels and the histogram had 128 buckets. These experiments are outlined as follows:

The first experiment sought to discover the effects of running the program with a varying number of processors. The program was run using 2^k processors for $k = 1, \ldots, 13$. Both the instruction execution time and network delay were set for a nominal 1 cycle. The results of this experiment are shown in Fig. 6. The performance of the program improves steadily as the number of processors is increased until 128 processors. Beyond 128 processors, the performance improvement is severely diminished. Figure 7 reveals the reason for this behavior by showing a summary of the execution profile for the program over this same range. The reduced performance is due to two factors: phases three, four, and five cannot use more than 128 processors, thus putting a lower bound on the execution time; further, phases two and six require more iterations as the number of processors grows beyond 128. These effects do not quite overshadow the continued improvement of phases one and seven, but the net improvement is very small relative to the cost of doubling the number of processors.

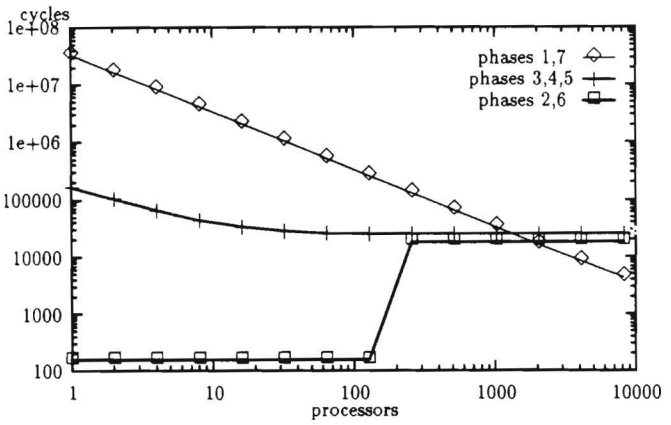

FIG. 7. Execution profile for an increasing number of processors.

The results of this first experiment show that phases three, four, and five benefit from having as many processors available as possible, up to a maximum of 128. This experiment further shows that phases one and seven benefit from as many processors as we have been able to simulate with RAW, but that the cost of phases two and six increase to the point where they diminish the benefit seen by phases one and seven.

The second experiment begins to factor the capabilities of the processors into the results. In the first experiment, we were not only doubling the number of processors, but also the amount of hardware in the system, so it is no surprise that the performance improved. In the second experiment, we use the same amount of hardware (see Section 4.2) in three different configurations. In configuration C1, the 128 processors each have a 32-bit data path, parallel multiply hardware, and some floating point support. The processors in configuration C2 also have 32-bit data paths, but have traded off parallel multiply and floating point support in order to obtain 256 processors. Configuration C3 has 8192 processors with one-bit data paths and simple boolean logic units much like those described in [8]. The results of the second experiment are shown in Figs. 8 and 9.

Figure 8 shows the overall execution time for the three configurations. This figure indicates that exploiting parallelism (C3) barely outweighs the advantage of faster hardware (C2) for this program. The profile summarized in Fig. 9 shows that phases one and seven dominate the execution in all three configurations. At the same time, it is clear that phases three, four, and five execute significantly faster in configuration C1. We surmise that a processor reconfigurable architecture would be able to take advantage of this fact to produce an improved speedup. Such an architecture would be capable of performing

phases one and seven in configuration C3 and phases three, four, and five in configuration C1.

A concern that arises is the fact that data objects used in configuration C1 will be computed in configuration C3, and likewise data objects used in configuration C1 (during phase 7) will be computed in configuration C1. The potential for a problem arises because configuration C1 and configuration C3 have a different data path width and each assumes that data objects can be accessed from the memory in a different format (see Appendix A for a discussion of bit-serial versus bit-parallel computation). Processor reconfigurable architectures such as those presented in [11, 21] require a software conversion of data items when moving from one configuration to another. In [20] Sandon proposes that processor reconfigurable architectures include reconfigurable memory modules that allow data access in multiple formats without the cost of a conversion. An implementation of such a memory is discussed in [15, 16]. Providing such a capability in hardware would certainly provide a performance improvement, but it would also represent a significant cost. Therefore, it is important to consider the performance impact of such a feature.

The third experiment simulates a reconfigurable architecture that is capable of assuming configurations C1 and C3 both with and without reconfigurable memory modules. The results of the third experiment are shown in Fig. 10. The first bar (Static) shows the time for configuration C3 from Experiment 2 as a reference. The second bar (Convert) is the time for an architecture that allows its PEs to be reconfigured dynamically but requires the memory to be reformatted in software for bit-serial and bit-parallel accesses. The third bar (No Convert) is for a reconfigurable architecture that employs a reconfigurable memory that allows data items to be accessed in either bit-serial or bit-parallel mode without a software conversion. For both of these reconfigurable architectures,

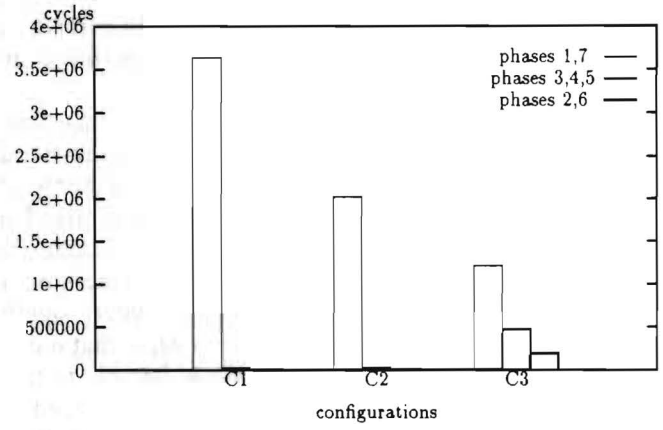

FIG. 9. Execution profile for three processor configurations.

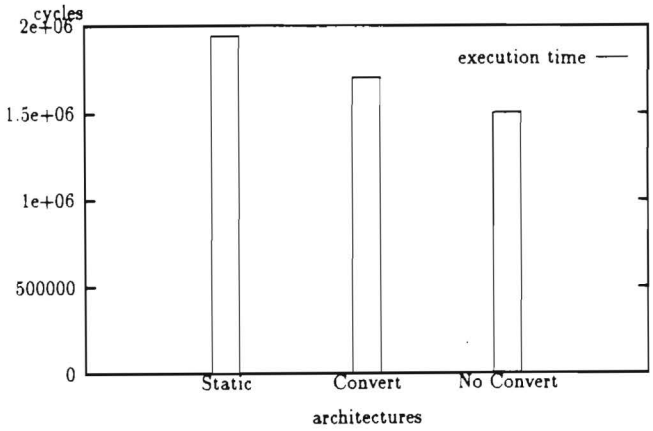

FIG. 10. Execution time for processor reconfigurable architectures.

Phases 1, 2, 6, and 7 are performed in configuration C3 and phases 3, 4, and 5 are performed in configuration C1. Figure 10 shows that processor reconfiguration provides a performance improvement, even if there is a conversion cost, and that the improvement is better if a conversion is NOT needed. The No Convert architecture achieves a 20% execution time reduction over the static architecture.

4.4. Putting the Results into the Proper Perspective

The results presented in the previous section indicate that an architecture such as configuration C3 may be a good choice for the histogram equalization program and that an architecture capable of reconfiguring into both C3 and C1 may be even better (by about 20%). However, this result is based on just one aspect of architectural reconfigurability. To continue with the iterative nature of our methodology (steps 3 and 4 in Section 1) we would then examine the other two dimensions of architectural reconfigurability, both individually and in tandem, with respect to this specific application. At this point, we would have arrived at architectural features that are well suited to the histogram equalization program. This process would provide a single data point in our study and would be repeated for all the programs in our target set. Presumably, other programs in the set would yield different results. Finally, we would arrive at a complex multidimensional set of experimental data which has to be optimized to give the best performance across the entire target set. This optimization will bring us to step 5 of the methodology. This step would involve designing features of the optimized architecture and refining the ADC models as a result of the design decisions. The later steps in the methodology would then apply these refined models to the benchmark suite to finally arrive at a specific design that can be carried through to implementation.

The experiments and the discussion presented in this section are intended to motivate the reader on how the methodology is carried out using RAW. We believe that following this methodology would result in realizing the potentials of architectural reconfigurability.

5. CONCLUSION

Reconfigurable architectures may provide an improvement in performance for a number of computation intensive application areas by allowing the architecture to adapt itself to the needs of the application, rather than requiring the application to adapt to the architecture. Our work takes a new step in studying this phenomenon by applying an empirical approach to the design of reconfigurable architectures. This approach utilizes an architectural analysis tool, the RAW which allows the designer to experimentally measure the performance implications of the different kinds of reconfigurability through simulation of target applications. We have built a prototype of this workbench, GT-RAW, and are using it to study processor reconfiguration for image understanding applications. We have presented an example of the experiments we are running and are currently working to refine these experiments to study the utility of specific hardware designs.

There are a number of interesting related issues that are worth pursuing. Among these are the potential for using the data generated by GT-RAW to drive the parallelization process, the theoretical aspects of the interaction of software and architecture, and better user interface designs for the visualization of the algorithm execution.

APPENDIX: BIT-SERIAL AND BIT-PARALLEL MODES

Typical SIMD architectures have one memory module per PE which is the same width as the PE's data path. Machines with one-bit PEs such as the Connection Machine store data values in a single memory module with each bit at consecutive addresses (Fig. 11). In this format, the bits of an operand are fetched one at a time and operated with the bits of another operand to produce result bits, which, in turn are stored one at a time back into the memory. On the other hand, processor reconfigurable architectures operating in a bit-parallel mode store their data values across several memory modules with each bit stored at the same address of a different module (Fig. 12). Note that "P" in the diagrams indicates a single one-bit processor. In bit-serial mode, there is one P per PE, and in bit parallel mode there are several Ps per PE. This allows the operands to be fetched and results to be stored each in a single cycle by emulating a single memory module with a wider data path. Problems arise

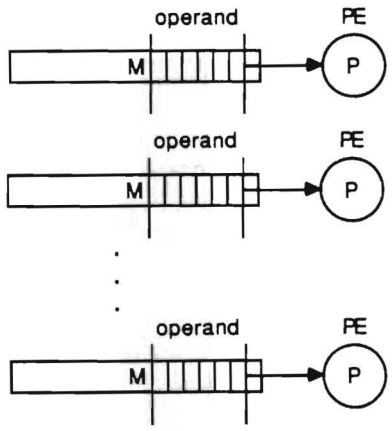

FIG. 11. Bit-serial processor.

when a processor reconfigurable architecture operates in both bit-serial mode and bit-parallel mode with the same data. If the data is stored in the bit-serial format, the bit-parallel PEs can still access it, but only one bit per cycle because the data is stored in a single module with a one-bit data path. This greatly diminishes the advantage of bit-parallel PEs. On the other hand, if the data is stored in the bit-parallel format, it can be accessed by having nearby PEs pass their corresponding bits to the correct PE, but only one PE can access its data at a time, which negates the advantage of having multiple PEs.

To overcome this problem, one of two options must be employed. First, the data can be reformatted via a software procedure. Second, the architecture can employ reconfigurable memory modules that allow the data to be accessed in either mode without being reformatted. Reconfigurable memory modules allow all of the bits of an operand to be accessed in one cycle and provide a means for routing the bits to the array of PEs for bit-parallel processing (Fig. 13). Alternatively, one bit from each of several operands can be selected and routed to each PE as in a normal bit-serial processor (Fig. 11). This is achieved by widening the data path of each memory module to the full word width and providing hardware to

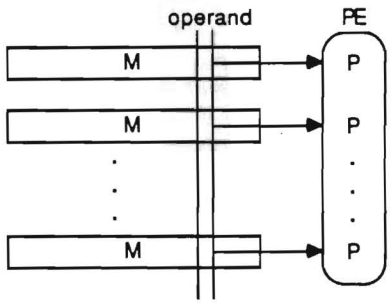

FIG. 12. Bit-parallel processor.

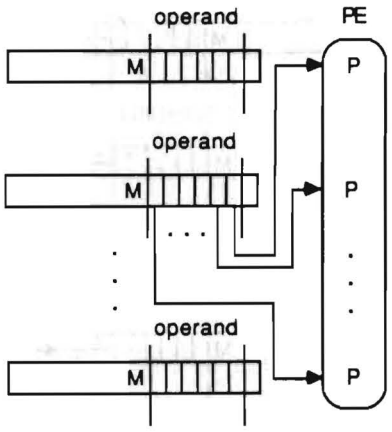

FIG. 13. Reconfigurable memory in bit-parallel mode.

handle addressing in each mode. The performance impact of the software approach versus the hardware approach depends on the characteristic of the specific application.

ACKNOWLEDGMENTS

This research is funded in part by an NSF PYI award MIPS-9058430 and an external research program award from Digital Equipment Corporation. We thank the referees for insightful comments that significantly helped to improve the presentation of this work. We also extend our appreciation to Professor Viktor Prasanna for the quick turnaround in getting this paper reviewed.

REFERENCES

1. Bronson, E. C., Casavant, T. L., and Siegel, L. J. Experimental application-driven architecture analysis of an SIMD/MIMD parallel processing system. *IEEE Trans. Parallel Distrib. Systems* **1** No. 2 (April 1990), 195–205.
2. Chen, S. C. Large-scale and high-speed multiprocessor system for scientific applications: Cray X-MP series. In *Proceedings, NATO Adv. Res. Workshop on High-Speed Computing, Julich, Germany, 1983*. Springer-Verlag, Berlin/New York.
3. *DARPA 1987 IU Workshop Proceedings, 1987.*
4. Enslow, P. H., Jr. Multiprocessor organization—A survey. *Comput. Surveys* **9,** No. 1 (1977), 103–129.
5. Feng, T. Y. Some characteristics of associative/parallel processing. In *Proceedings, 1927 Sagamore Computer Conf., Syracuse University, Syracuse, 1972,* pp. 5–16.
6. Flynn, M. J. Very high-speed computing systems. In *Proc. IEEE* **54** (1966), 1901–1909.
7. Handler, W. The impact of classification schemes on computer architecture. In *Proceedings, Int. IEEE Conf. on Parallel Processing, New York, 1977,* pp. 7–15.
8. Hillis, W. D. *The Connection Machine.* MIT Press, Cambridge, MA, 1985.
9. Hillis, W. D., and Steele, G. L. Data parallel algorithms. *Comm. ACM* **29,** No. 12 (December 1986), 1170–1183.
10. Jamieson, L. H., Mueller, P. T., Jr., and Siegel, H. J. FFT algorithms for SIMD parallel processing systems. *J. Parallel Distrib. Comput.* **3** (1986), 48–71.
11. Kartashev, S. I., and Kartashev, S. P. A multicomputer system with dynamic architecture. *IEEE Trans. Comput.* **C-28,** No. 10 (October 1979), 704–721.
12. Kung, H. T. The structure of parallel algorithms. *Adv. in Comput.* **19** (1980), 65–112.
13. Li, H., and Maresca, M. Polymorphics-torus network. In *Proceedings, Int. IEEE Conf. on Parallel Processing, New York, 1987.*
14. Ligon, W. B. *An Empirical Analysis of Reconfigurable Architectures.* Doctoral thesis, Georgia Institute of Technology, in preparation.
15. Ligon, W. B., and Ramachandran, U. *A Reconfigurable Supercomputer Architecture.* Technical Report GIT-ICS-89/13, Georgia Institute of Technology, February 1989.
16. Ligon, W. B., and Ramachandran, U. Exploration of reconfigurable architectures: An empirical approach. In *Proceedings, 3rd. IEEE, Symposium on the Frontiers of Massively Parallel Computation, New York, October 1990,* pp. 205–214.
17. MacDougall, M. H. *Simulating Computer Systems: Techniques and Tools.* MIT Press, Cambridge, MA, 1987.
18. Miller, R., Prasanna Kumar, V. K., Reisis, D., and Stout, Q. F. Meshes with reconfigurable buses. In *Proceedings, Conf. Advanced Research in VLSI, 1988.*
19. Patterson, D. A. The case for the reduced instruction set computer. *Comput. Architecture News* **8,** No. 6 (October 1980), 25–33.
20. Sandon, P. A. A pyramid implementation using a reconfigurable array of processors. In *Proceedings, 1985 Conf. on Computer Architecture for Pattern Analysis and Image Database Management, 1985,* pp. 112–118.
21. Sejnowski, M. C., Upchurch, E. T., Kapur, R. N., Charlu, D. P. S., and Lipovski, G. J. An overview of the Texas Reconfigurable Array Computer. In *Proceedings, 1980 AFIPS Nat. Comput. Conf., Arlington, May 1980,* pp. 631–641. AFIPS Press, Reston, VA.
22. Siegel, H. J., Siegel, L. J., Kemmerer, F. C., Mueller, P. T., Jr., Smalley, H. E., Jr., and Smith, S. D. PASM: A partitionable SIMD/MIMD system for image processing and pattern recognition. *IEEE Trans. Comput.* **C-30,** No. 12 (December 1981), 934–947.
23. Siegel, L. J. Image processing on a partitionable SIMD machine. In *Languages and Architectures for Image Processing, 1981,* pp. 293–300.
24. Snyder, L. Introduction to the configurable, highly parallel computer. *IEEE Comput.* (January 1982), 47–56.
25. Snyder, L. A taxonomy of synchronous parallel machines. In *Proc. Int. Conf. on Parallel Processing, August 1988,* pp. 281–285.
26. Weams, C., Hanson, A., Riseman, E., and Rosenfeld, A. An integrated image understanding benchmark. Technical report, University of Massachusetts, in preparation.
27. Weams, C. C., Levitan, S. P., Hanson, A. R., Riseman, E. M., Nash, J. G., and Shu, D. B. The image understanding architecture. *Int. J. Comput. Vision* **2** (1988), 251–282.

WALTER B. LIGON III received **B.S., M.S., and Ph.D.** degrees in Computer Science from Georgia Institute of Technology. He is currently an assistant professor in the Department of Electrical and Computer Engineering at Clemson University. His research interests are in computer architecture, computer vision, parallel programming, compilers, and software engineering. The projects he is currently associated

with are RAW (reconfigurable architecture workbench) and CALVIN (collaborative assignment laboratory/virtual interaction).

UMAKISHORE RAMACHANDRAN received his Ph.D. in Computer Science from the University of Wisconsin, Madison, in 1986. Since then he has been with the College of Computing at Georgia Institute of Technology, where he is currently an Associate Professor. His primary research interests are in computer architecture and operating systems. The projects he is currently associated with are TASS (a top-down approach to scalability study of parallel architectures), Beehive (a scalable shared memory multiprocessor), RAW (a reconfigurable architecture workbench), and Clouds (an object-based distributed operating system). He was the recipient of an NSF Presidential Young Investigator Award in 1990.

Received November 7, 1990; revised December 10, 1991; accepted May 7, 1992

To appear in *Proceedings of the First International Symposium on High Performance Computer Architecture*, January 1995.

Abstracting Network Characteristics and Locality Properties of Parallel Systems*

Anand Sivasubramaniam *Aman Singla* *Umakishore Ramachandran* *H. Venkateswaran*

College of Computing
Georgia Institute of Technology
Atlanta, GA 30332-0280
{*anand,aman,rama,venkat*}*@cc.gatech.edu*

Abstract

Abstracting features of parallel systems is a technique that has been traditionally used in theoretical and analytical models for program development and performance evaluation. In this paper, we explore the use of abstractions in execution-driven simulators in order to speed up simulation. In particular, we evaluate abstractions for the interconnection network and locality properties of parallel systems in the context of simulating cache-coherent shared memory (CC-NUMA) multiprocessors. We use the recently proposed LogP model to abstract the network. We abstract locality by modeling a cache at each processing node in the system which is maintained coherent, without modeling the overheads associated with coherence maintenance. Such an abstraction tries to capture the true communication characteristics of the application without modeling any hardware induced artifacts. Using a suite of applications and three network topologies simulated on a novel simulation platform, we show that the latency overhead modeled by LogP is fairly accurate. On the other hand, the contention overhead can become pessimistic when the applications display sufficient communication locality. Our abstraction for data locality closely models the behavior of the target system over the chosen range of applications. The simulation model which incorporated these abstractions was around 250-300% faster than the simulation of the target machine.

1 Motivation

Performance analysis of parallel systems[1] is complex due to the numerous degrees of freedom that they exhibit. Developing algorithms for parallel architectures is also hard if one has to grapple with all parallel system artifacts. Abstracting features of parallel systems is a technique often employed to address both of these issues. For instance, abstracting parallel machines by theoretical models like the PRAM [14] has facilitated algorithm development and analysis. Such models try to hide hardware details from the programmer, providing a simplified view of the machine. Similarly, analytical models used in performance evaluation abstract complex system interactions with simple mathematical formulae, parameterized by a limited number of degrees of freedom that are tractable.

There is a growing awareness for evaluating parallel systems using applications due to the dynamic nature of the interaction between applications and architectures. Execution-driven simulation is becoming an increasingly popular vehicle for performance prediction because of its ability to accurately capture such complex interactions in parallel systems [25, 22]. However, simulating every artifact of a parallel system places tremendous requirements on resource usage, both in terms of space and time. A sufficiently abstract simulation model which does not compromise on accuracy can help in easing this problem. Hence, it is interesting to investigate the use of abstractions for speeding up execution-driven simulations which is the focus of this study. In particular, we address the issues of abstracting the *interconnection network* and *locality* properties of parallel systems.

Interprocess communication (both explicit via messages or implicit via shared memory), and locality are two main characteristics of a parallel application. The interconnection network is the hardware artifact that facilitates communication and an interesting question to be addressed is if it can be abstracted without sacrificing the accuracy of the performance analysis. Since latency and contention are the two key attributes of an interconnection network that impacts the application performance, any model for the network should capture these two attributes. There are two aspects to locality as seen from an application: communication locality and data locality. The properties of the interconnection network determine the extent to which communication locality is exploited. In this sense, the abstraction for the interconnection network subsumes the effect of communication locality. Exploiting data locality is facilitated either by private caches in shared memory multiprocessors, or local memories in distributed memory machines. Focusing only on shared memory multiprocessors, an important question that arises is to what extent caches can be abstracted and still be useful in program design and performance prediction. It is common for most shared memory multiprocessors to have coherent caches, and the cache plays an important role in reducing network traffic. Hence, it is clear that any abstraction of such a machine has to model a cache at each node. On the other hand, it is not apparent if a simple abstraction can accurately capture the important behavior of caches in reducing network traffic.

We explore these two issues in the context of simulating Cache Coherent Non-Uniform Memory Access (CC-NUMA) shared memory machines. For abstracting the interconnection network, we use the recently proposed *LogP* [11] model that incorporates the two defining characteristics of a network, namely, latency and contention. For abstracting the locality properties of a parallel system, we model a private cache at each processing node in the system to capture data locality. Note that the communication locality is subsumed in the abstraction for the interconnection

*This work has been funded in part by NSF grants MIPS-9058430 and MIPS-9200005, and an equipment grant from DEC.

[1]The term, parallel system, is used to denote an application-architecture combination.

network. Thus in the rest of the paper (unless explicitly stated otherwise) we use the term 'locality' to simply mean data locality. Shared memory machines with private caches usually employ a protocol to maintain coherence. With a diverse range of cache coherence protocols, it would become very specific if our abstraction were to model any particular protocol. Further, memory references (locality) are largely dictated by application characteristics and are relatively independent of cache coherence protocols. Hence, instead of modeling any particular protocol, we choose to maintain the caches coherent in our abstraction but do not model the overheads associated with maintaining the coherence. Such an abstraction would represent an ideal coherent cache that captures the true inherent locality in an application.

The study uses an execution-driven simulation framework which identifies, isolates, and quantifies the different overheads that arise in a parallel system. Using this framework, we simulate the execution of five parallel applications on three different machine characterizations: a *target* machine, a *LogP* machine and a *cLogP* machine. The target machine simulates the pertinent details of the hardware. The LogP machine does not model private caches at processing nodes, and abstracts the interconnection network using the LogP model. The cLogP machine abstracts the locality properties using the above mentioned scheme, and abstracts the interconnection network using the LogP model. To answer the first question regarding network abstraction, we compare the simulation of the target machine to the simulation of the cLogP machine. If the network overheads of the two simulations agree then we have shown that LogP is a good abstraction for the network. To answer the second question regarding locality abstraction, we compare the network traffic generated by the target and cLogP machines. If they agree, then it shows that our abstraction of the cache is sufficient to model locality. Incidentally, the difference in results between the target and LogP simulations would quantify the impact of locality on performance. If the difference is substantial (as we would expect it to be), then it shows that locality cannot be abstracted out entirely.

Our results show that the latency overhead modeled by LogP is fairly accurate. On the other hand, the contention overhead modeled by LogP can become pessimistic for some applications due to failure of the model to capture communication locality. The pessimism gets amplified as we move to networks with lower connectivity. With regard to the data locality question, results show that our ideal cache, which does not model any coherence protocol overheads, is a good abstraction for capturing locality over the chosen range of applications. Abstracting the network and cache behavior also helped lower the cost of simulation by a factor of 250-300%. Given that execution-driven simulations of real applications can take an inordinate amount of time (some of the simulations in this study take between 8-10 hours), this factor can represent a substantial saving in simulation time.

Section 2 addresses related work and section 3 gives details on the framework that has been used to conduct this study. We use a set of applications (Section 4) and a set of architectures (Section 5) as the basis to address these questions. Performance results are presented in Section 6 and a discussion of the implication of the results is given in Section 7. Section 8 presents concluding remarks.

2 Related Work

Abstracting machine characteristics via a few simple parameters have been traditionally addressed by theoretical models of computation. The PRAM model assumes conflict-free accesses to shared memory (assigning unit cost for memory accesses) and zero cost for synchronization. The PRAM model has been augmented with additional parameters to account for memory access latency [4], memory access conflicts [5], and cost of synchronization [15, 9]. The Bulk Synchronous Parallel (BSP) model [28] and the LogP model [11] are departures from the PRAM models, and attempt to realistically bridge the gap between theory and practice. Similarly, considerable effort has been expended in the area of performance evaluation in developing simple analytical abstractions to model the complex behavior of parallel systems. For instance, Agarwal [2] and Dally [12] develop mathematical models for abstracting the network and studying network properties. Patel [19] analyzes the impact of caches on multiprocessor performance. But many of these models make simplifying assumptions about the hardware and/or the applications, restricting their ability to model the behavior of real parallel systems.

Execution-driven simulation is becoming increasingly popular for capturing the dynamic behavior of parallel systems [25, 8, 10, 13, 20]. Some of these simulators have abstracted out the instruction-set of the processors, since a detailed simulation of the instruction-set is not likely to contribute significantly to the performance analysis of parallel systems. Researchers have tried to use other abstractions for the workload as well as the simulated hardware in order to speed up the simulation. In [29] a Petri net model is used for the application and the hardware. Mehra et al. [17] use application knowledge in abstracting out phases of the execution.

The issue of locality has been well investigated in the architecture community. Several studies [3, 16] have explored hardware facilities that would help exploit locality in applications, and have clearly illustrated the use of caches in reducing network traffic. There have also been application-driven studies which try to synthesize cache requirements from the application viewpoint. For instance, Gupta et al. [21] show that a small-sized cache of around 64KB can accommodate the important working set of many applications. Similarly, Wood et al. [30] show that the performance of a suite of applications is not very sensitive to different cache coherence protocols. But from the performance evaluation viewpoint, there has been little work done in developing suitable abstractions for modeling the locality properties of a parallel system which can be used in an execution-driven simulator.

3 The Framework

In this section, we present the framework that is used to answer the questions raised earlier. We give details of the three simulated machine characterizations and the simulator that has been used in this study.

The "target" machine is a CC-NUMA shared memory multiprocessor. Each node in the system has a piece of the globally shared memory and a private cache that is maintained sequentially consistent using an invalidation-based (Berkeley protocol) fully-mapped directory-based cache coherence scheme. The pertinent hardware features of the interconnection network and coherence maintenance are simulated, and section 5 gives further details of this machine.

3.1 The LogP Machine

The LogP model proposed by Culler et al. [11] assumes a collection of processing nodes executing asynchronously, communicating with each other by small fixed-size messages incurring

constant latencies on a network with a finite bandwidth. The model defines the following set of parameters that are independent of network topology:

- L: the *latency*, is the maximum time spent in the network by a message from a source to any destination.
- o: the *overhead*, is the time spent by a processor in the transmission/reception of a message.
- g: the communication *gap*, is the minimum time interval between consecutive message transmissions/receptions from/to a given processor.
- P: is the number of processors in the system.

The L-parameter captures the actual network transmission time for a message in the absence of any contention, while the g-parameter corresponds to the available per-processor bandwidth. By ensuring that a processor does not exceed the per-processor bandwidth of the network (by maintaining a gap of at least g between consecutive transmissions/receptions), a message is not likely to encounter contention.

We use the L and g parameters of the model to abstract the network in the simulator. Since we are considering a shared memory platform (where the 'message overhead' is incurred in the hardware) the contribution of the o-parameter is insignificant compared to L and g, and we do not discuss it in the rest of this paper. Our LogP machine is thus a collection of processors, each with a piece of the globally shared memory, connected by a network which is abstracted by the L and g parameters. Due to the absence of caches, any non-local memory reference would need to traverse the network as in a NUMA machine like the Butterfly GP-1000. In our simulation of this machine, each message in the network incurs a latency L that accounts for the actual transmission time of the message. In addition, a message may incur a waiting time at the sending/receiving node as dictated by the g parameter. For instance, when a node tries to send a message, it is ensured that at least g time units have elapsed since the last network access at that node. If not the message is delayed appropriately. A similar delay may be experienced by the message at the receiving node. These delays are expected to model the contention that such a message would encounter on an actual network.

3.2 The cLogP Machine

The LogP machine augmented with an abstraction for a cache at each processing node is referred to as a cLogP machine. A network access is thus incurred only when the memory request cannot be satisfied by the cache or local memory. The caches are maintained coherent conforming to a sequentially consistent memory model. With a diverse number of cache coherence protocols that exist, it would become very specific if cLogP were to model any particular protocol. Further, the purpose of the cLogP model is to verify if a simple minded abstraction for the cache can closely model the behavior of the corresponding "target" machine without having to model the details of any specific cache coherence protocol, since it is not the intent of this study to compare different cache coherence protocols. In the cLogP model the caches are maintained consistent using an invalidation based protocol (Berkeley protocol), but the overhead for maintaining the coherence is not modeled. For instance, consider the case where a block is present in a valid state in the caches of two processors. When a processor writes into the block, an invalidation message would be generated on the "target" machine, but there would not be any network access for this operation on the cLogP machine. The block would still change to 'invalid' state on both machines after this operation. A read by the other processor after this operation, would incur a network access on both machines. cLogP thus tries to capture the true communication characteristics of the application, ignoring overheads that may have been induced by hardware artifacts, representing the minimum number of network messages that any coherence protocol may hope to achieve. If the network accesses incurred in the cLogP model are significantly lower than the accesses on the "target" machine, then we would need to make our cLogP abstraction more realistic. But our results (to be presented in section 6) show that the two agree very closely over the chosen range of applications, confirming our choice for the cLogP abstraction in this study. Furthermore, if the target machine implements a fancier invalidation-based cache coherence protocol (which would reduce the network accesses even further), that would only enhance the agreement between the results for the cLogP and target machines.

3.3 SPASM

In this study, we use an execution-driven simulator called SPASM (Simulator for Parallel Architectural Scalability Measurements) that enables us to accurately model the behavior of applications on a number of simulated hardware platforms. SPASM has been written using CSIM [18], a process oriented sequential simulation package, and currently runs on SPARCstations. The input to the simulator are parallel applications written in C. These programs are pre-processed (to label shared memory accesses), the compiled assembly code is augmented with cycle counting instructions, and the assembled binary is linked with the simulator code. As with other recent simulators [8, 13, 10, 20], bulk of the instructions is executed at the speed of the native processor (the SPARC in this case) and only instructions (such as LOADs and STOREs on a shared memory platform or SENDs and RECEIVEs on a message-passing platform) that may potentially involve a network access are simulated. The reader is referred to [27, 25] for a detailed description of SPASM where we illustrated its use in studying the scalability of a number of parallel applications on different shared memory [25] and message-passing [27] platforms. The input parameters that may be specified to SPASM are the number of processors, the CPU clock speed, the network topology, the link bandwidth and switching delays.

SPASM provides a wide range of statistical information about the execution of the program. It gives the *total time* (simulated time) which is the maximum of the running times of the individual parallel processors. This is the time that would be taken by an execution of the parallel program on the target parallel machine. The profiling capabilities of SPASM (outlined in [25]) provide a novel isolation and quantification of different overheads in a parallel system that contribute to the performance of the parallel system. These overheads may be broadly separated into a purely algorithmic component, and an interaction component arising from the interaction of the algorithm with the architecture. The algorithmic overhead arises from factors such as the serial part and work-imbalance in the algorithm, and is captured by the *ideal time* metric provided by SPASM. Ideal time is the time taken by the parallel program to execute on an ideal machine such as the PRAM [31]. This metric includes the algorithmic overheads but does not include any overheads arising from architectural limitations. Of the interaction component, the *latency* and *contention* introduced by network limitations are the important overheads

that are of relevance to this study. The time that a message would have taken for transmission in a contention free environment is charged to the latency overhead, while the rest of the time spent by a message in the network waiting for links to become free is charged to the contention overhead.

The separation of overheads provided by SPASM plays a crucial role in this study. For instance, even in cases where the overall execution times may agree, the latency and contention overheads provided by SPASM may be used to validate the corresponding estimates provided by the L and g parameters in LogP. Similarly, the latency overhead (which is an indication of the number of network messages) in the target and cLogP machine may be used to validate our locality abstraction in the cLogP model. In related studies, we have illustrated the importance of separating parallel system overheads in scalability studies of parallel systems [25], identifying parallel system (both algorithmic and architectural) bottlenecks [25], and synthesizing architectural requirements from an application viewpoint [26].

4 Application Characteristics

Three of the applications (EP, IS and CG) used in this study are from the NAS parallel benchmark suite [7]; CHOLESKY is from the SPLASH benchmark suite [23]; and FFT is the well-known Fast Fourier Transform algorithm. EP and FFT are well-structured applications with regular communication patterns determinable at compile-time, with the difference that EP has a higher computation to communication ratio. IS also has a regular communication pattern, but in addition it uses locks for mutual exclusion during the execution. CG and CHOLESKY are different from the other applications in that their communication patterns are not regular (both use sparse matrices) and cannot be determined at compile time. While a certain number of rows of the matrix in CG is assigned to a processor at compile time (static scheduling), CHOLESKY uses a dynamically maintained queue of runnable tasks. Further details of the applications are given in [24].

5 Architectural Characteristics

Since uniprocessor architecture is getting standardized with the advent of RISC technology, we fix most of the processor characteristics by using a 33 MHz SPARC chip as the baseline for each processor in a parallel system. Such an assumption enables us to make a fair comparison of the relative merits of the interesting parallel architectural characteristics across different platforms.

The study is conducted for the following interconnection topologies: the *fully connected network*, the *binary hypercube* and the *2-D mesh*. All three networks use serial (1-bit wide) unidirectional links with a link bandwidth of 20 MBytes/sec. The fully connected network models two links (one in each direction) between every pair of processors in the system. The cube platform connects the processors in a binary hypercube topology. Each edge of the cube has a link in each direction. The 2-D mesh resembles the Intel Touchstone Delta system. Links in the North, South, East and West directions, enable a processor in the middle of the mesh to communicate with its four immediate neighbors. Processors at corners and along an edge have only two and three neighbors respectively. Equal number of rows and columns is assumed when the number of processors is an even power of 2. Otherwise, the number of columns is twice the number of rows (we restrict the number of processors to a power of 2 in this study). Messages are circuit-switched and use a wormhole routing strategy. Message-sizes can vary upto 32 bytes. The switching delay is assumed to be negligible compared to the transmission time and we ignore it in this study.

Each node in the simulated CC-NUMA hierarchy is assumed to have a sufficiently large piece of the globally shared memory such that for the applications considered, the data-set assigned to each processor fits entirely in its portion of shared memory. The private cache modeled in the "target" and the "cLogP" machines is a 2-way set-associative cache (64KBytes with 32 byte blocks) that is maintained sequentially consistent using an invalidation-based (Berkeley protocol) fully-mapped directory-based cache coherence scheme. The L parameter for a message on the LogP and cLogP models is chosen to be 1.6 microseconds assuming 32-byte messages and a link bandwidth of 20 MBytes/sec. Similar to the method used in [11], the g parameter is calculated using the cross-section bandwidth available per processor for each of the above network configurations. The resulting g parameters for the full, cube and mesh networks are respectively, $3.2/p$, 1.6 and $0.8 * p_x$ microseconds (where p is the number of processors and p_x is the number of columns in the mesh).

6 Performance Results

The simulation results for the five parallel applications on the target machine, and the LogP and cLogP models of the machine are discussed in this section. The results presented include the execution times, latency overheads, and contention overheads for the execution of the applications on the three network topologies. We confine our discussion to the specific results that are relevant to the questions raised earlier. EP, FFT, and IS are applications with statically determinable memory reference patterns (see the appendix). Thus, in implementing these applications we ensured that the amount of communication (due to non-local references) is minimized. On the other hand, CG and CHOLESKY preclude any such optimization owing to their dynamic memory reference patterns.

6.1 Abstracting the Network

For answering the question related to network abstractions, we compare the results obtained using the cLogP and the target machines. From Figures 1, 2, 3, 4, and 5, we observe that the latency overhead curves for the cLogP machine display a trend (shape of the curve) very similar to the target machine thus validating the use of the L-parameter of the LogP model for abstracting the network latency. For the chosen parallel systems, there is negligible difference in latency overhead across network platforms since the size of the messages and transmission time dominate over the number of hops traversed. Since LogP model abstracts the network latency independent of the topology the other two network platforms (cube and mesh) also display a similar agreement between the results for the cLogP and target machines. Therefore, we show the results for only the fully connected network. Despite this similar trend, there is a difference in the absolute values for the latency overheads. cLogP models L as the time taken for a cache-block (32 bytes) transfer. But some messages may actually be shorter making L pessimistic with respect to the target machine. On the other hand, cLogP does not model coherence traffic thereby incurring fewer network messages than the target machine, which can have the effect of making L more optimistic. The impact of these two counter-acting effects on the

overall performance depends on the application characteristics. The pessimism is responsible for cLogP displaying a higher latency overhead than the target machine for FFT (Figure 1) and CG (Figure 2) since there is very little coherence related activity in these two applications; while the optimism favors cLogP in IS (Figure 4) and CHOLESKY (Figure 5) where coherence related activity is more prevalent. However, it should be noted that these differences in absolute values are quite small implying that the L parameter pretty closely models the latency attribute.

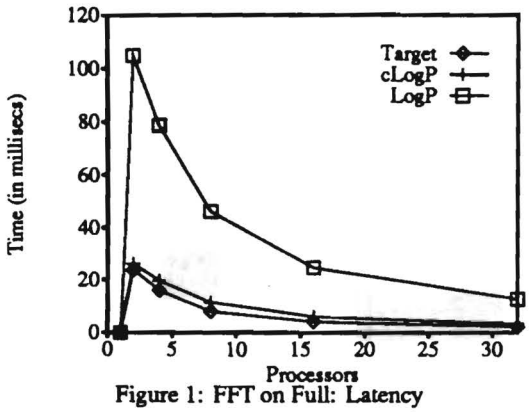

Figure 1: FFT on Full: Latency

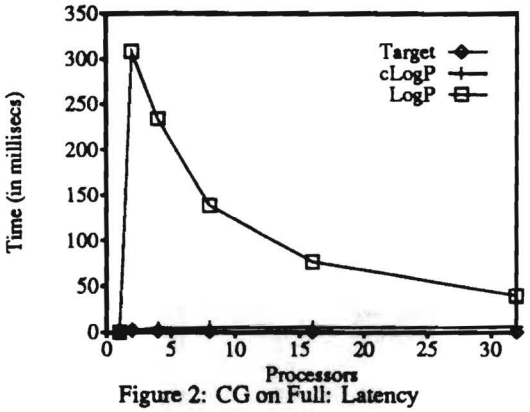

Figure 2: CG on Full: Latency

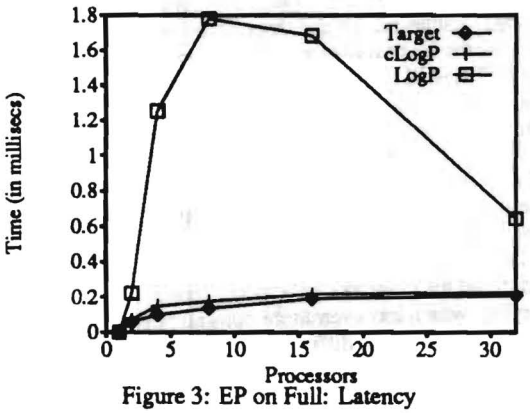

Figure 3: EP on Full: Latency

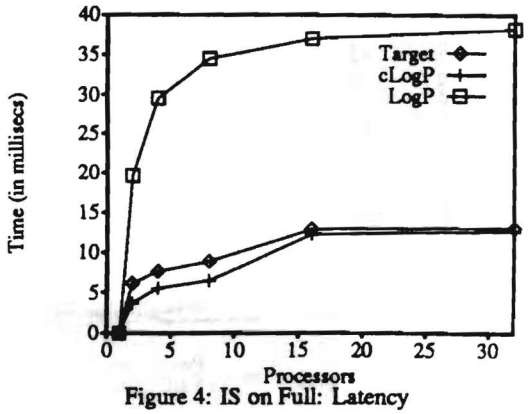

Figure 4: IS on Full: Latency

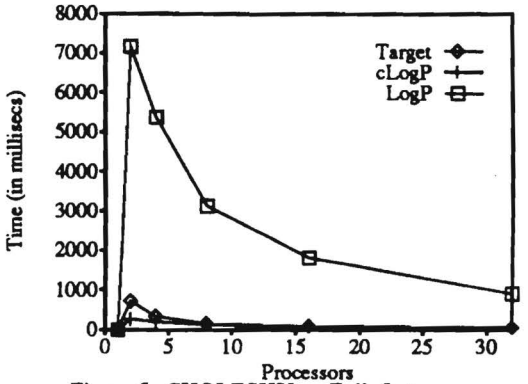

Figure 5: CHOLESKY on Full: Latency

Figures 6, 7, 8, and 9, show that the contention overhead curves for the cLogP machine display a trend (shape of the curves) similar to the target machine. But there is a difference in the absolute values. The g-parameter in cLogP is estimated using the bisection bandwidth of the network as suggested in [11]. Such an estimate assumes that every message in the system traverses the bisection and can become very pessimistic when the application displays sufficient communication locality [1, 2]. This pessimism increases as the connectivity of the network decreases (as can be seen in Figures 6 and 7) since the impact of communication locality increases. This pessimism is amplified further for applications such as EP that display a significant amount of communication locality. This effect can be seen in Figures 10 and 11 which show a significant disparity between the contention on the cLogP and target machines. In fact, this amplified effect changes the very trend of the cLogP contention curves compared to the target machine. These results indicate that the contention estimated by the g parameter can turn out to be pessimistic, especially when the application displays sufficient communication locality. Hence, we need to find a better parameter for estimating the contention overhead, or we would at least need to find a better way of estimating g that incorporates application characteristics.

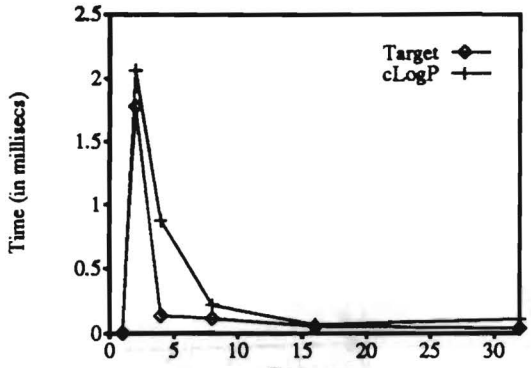

Figure 6: IS on Full: Contention

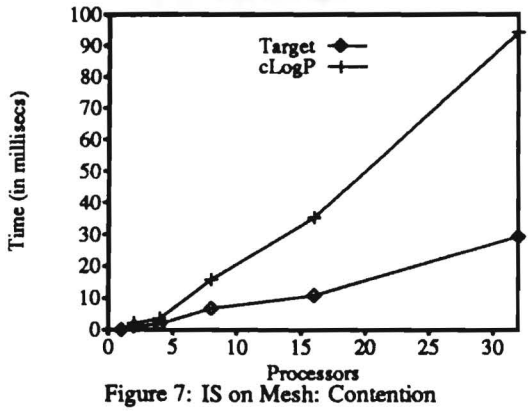

Figure 7: IS on Mesh: Contention

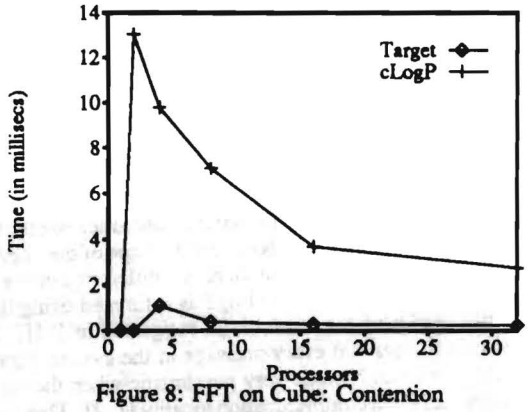

Figure 8: FFT on Cube: Contention

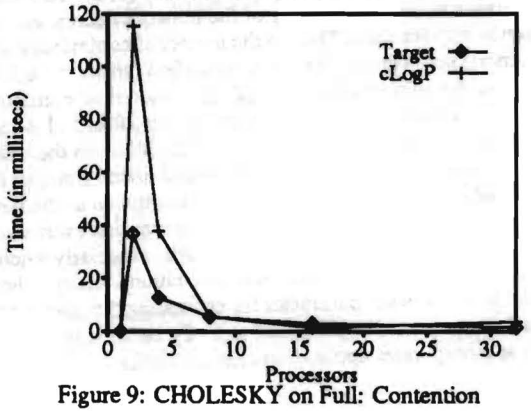

Figure 9: CHOLESKY on Full: Contention

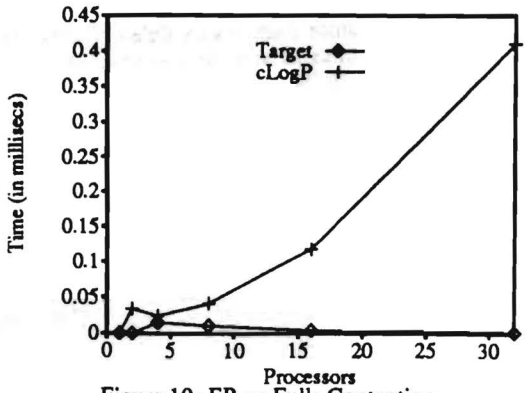

Figure 10: EP on Full: Contention

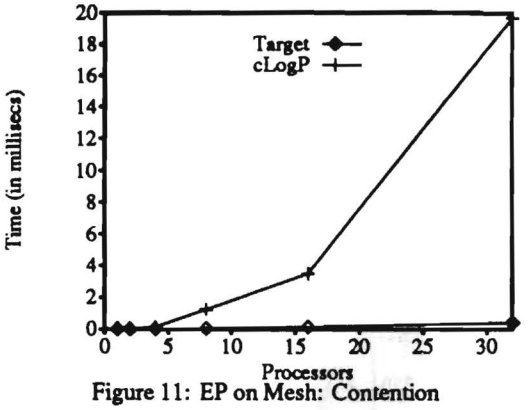

Figure 11: EP on Mesh: Contention

6.2 Abstracting Locality

Recall that our abstraction for locality attempts to capture the inherent data locality in an application. The number of messages generated on the network due to non-local references in an application is the same regardless of the network topology. Even though the number of messages stays the same, the contention is expected to increase when the connectivity in the network decreases. Therefore, the impact of locality is expected to be more for a cube network compared to a full; and for a mesh compared to a cube.

The impact of ignoring locality in a performance model is illustrated by comparing the execution time curves for the LogP and cLogP machines. Of the three static applications (EP, FFT, IS), EP has the highest computation to communication ratio, followed by FFT, and IS. Since the amount of communication in EP is minimal, there is agreement in the results for the LogP, the cLogP, and the target machines (Figure 12), regardless of network topology. On the fully connected and cube networks there is little difference in the results for FFT as well, whereas for the mesh interconnect the results are different between LogP and cLogP (Figure 13). The difference is due to the fact that FFT has more communication compared to EP, and the effect of non-local references is amplified for networks with lower connectivity. For IS (see Figure 14), which has even more communication than FFT, there is a more pronounced difference between LogP and cLogP on all three networks. For applications like CG and CHOLESKY which exhibit dynamic communication behavior, the difference between LogP and cLogP curves is more significant (see Figures 15 and 16) since the LogP implementation cannot be optimized statically

to exploit locality. Further, as we move to networks with lower connectivity, the LogP execution curves for CG and CHOLESKY (Figures 17 and 18) do not even follow the shape of the cLogP execution curves. This significant deviation of LogP from cLogP execution is due to the amplified effect of the large amount of communication stemming from the increased contention in lower connectivity networks (see Figures 19 and 20).

Isolating the latency and contention overheads from the total execution time (see section 3) helps us identify and quantify locality effects. Figures 1, 2, and 3, illustrate some of these effects for FFT, CG, and EP respectively. During the communication phase in FFT, a processor reads consecutive data items from an array displaying spatial locality. In either the cLogP or the target machine, a cache-miss on the first data item brings in the whole cache block (which is 4 data items). On the other hand, in the LogP machine all four data items result in network accesses. Thus FFT on the LogP machine incurs a latency (Figure 1) which is approximately four times that of the other two. Similarly, ignoring spatial and temporal locality in CG (Figure 2) results in a significant disparity for the latency overhead in the LogP machine compared to the other two. In EP, a processor waits on a condition variable to be signaled by another (see the appendix). For EP on a cLogP machine, only the first and last accesses to the condition variable use the network, while on the LogP machine a network access would be incurred for each reference to the condition variable as is reflected in Figure 3. Similarly, a test-test&set primitive [6], would behave like an ordinary test&set operation in the LogP machine thus resulting in an increase of network accesses. As can be seen in Figure 12, these effects do not impact the total execution time of EP since computation dominates for this particular application.

The above results confirm the well known fact that locality cannot be ignored in a performance prediction model or in program development. On the other hand, the results answer the more interesting question of whether the simple abstraction we have chosen for modeling locality in cLogP is adequate, or if we have to look for a more accurate model. cLogP does a fairly good job of modeling the cache behavior of the target machine. The above results clearly show that the execution curves of cLogP and the target machine are in close agreement across all application-architecture combinations. Further, the latency overhead curves (which are indicative of the number of messages exchanged between processors) of cLogP and the target machine are also in close agreement. This suggests that our simple abstraction for locality in cLogP, an ideal coherent cache with no overhead associated with coherence maintenance, is sufficient to model the locality properties over the chosen range of applications.

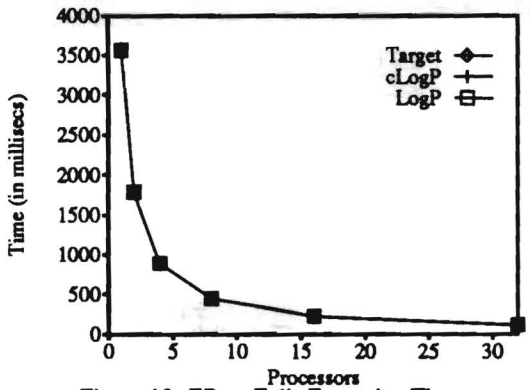

Figure 12: EP on Full: Execution Time

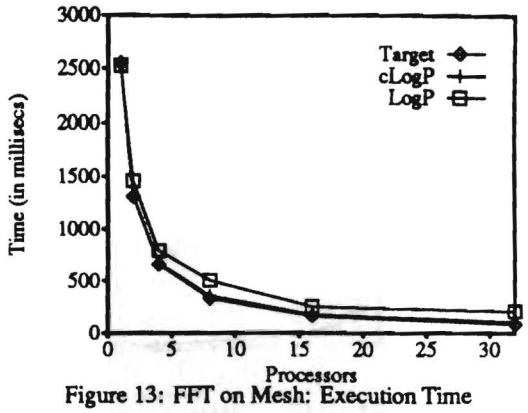

Figure 13: FFT on Mesh: Execution Time

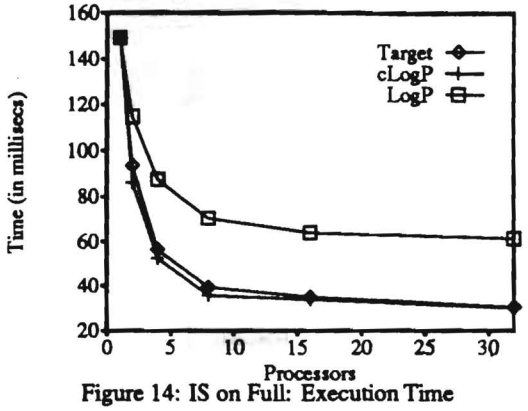

Figure 14: IS on Full: Execution Time

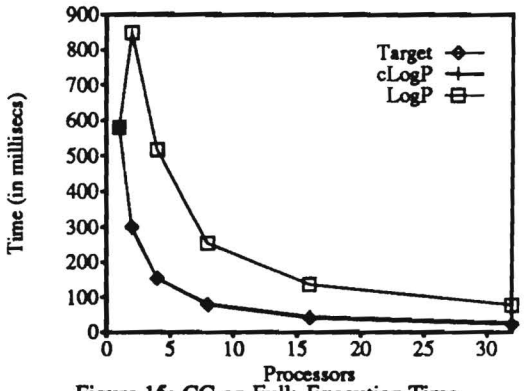

Figure 15: CG on Full: Execution Time

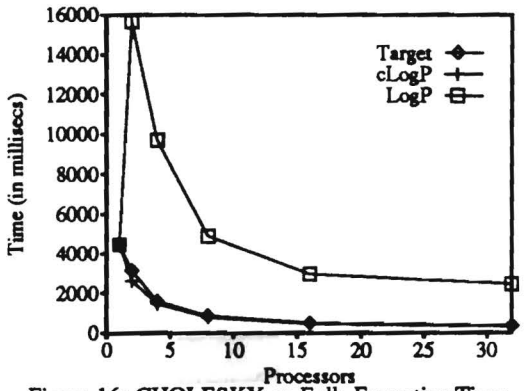

Figure 16: CHOLESKY on Full: Execution Time

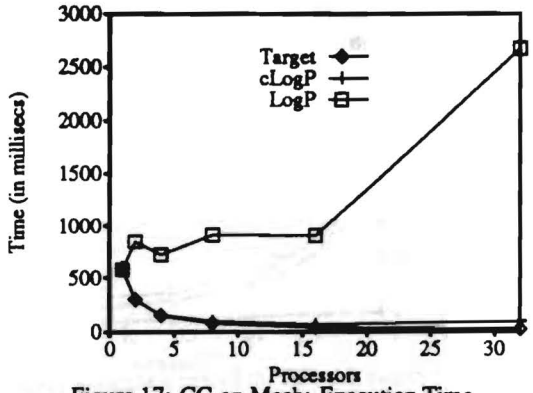

Figure 17: CG on Mesh: Execution Time

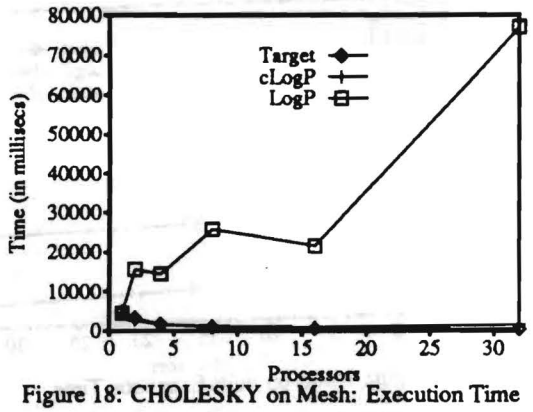

Figure 18: CHOLESKY on Mesh: Execution Time

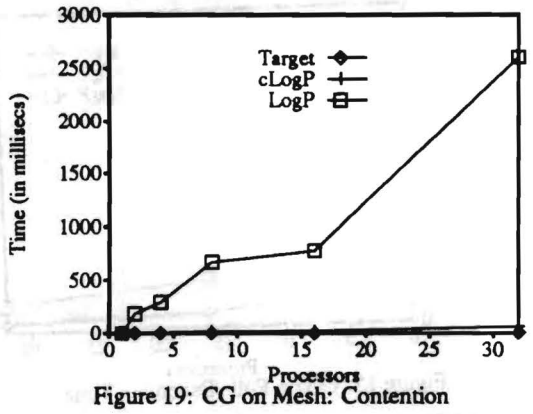

Figure 19: CG on Mesh: Contention

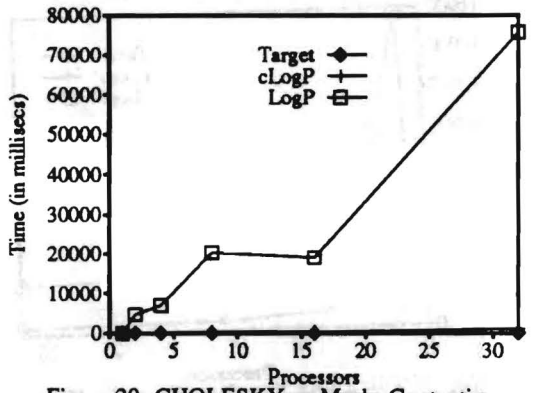

Figure 20: CHOLESKY on Mesh: Contention

7 Discussion

We considered the issues pertaining to abstracting network characteristics and locality in this study in the context of five parallel scientific applications with different characteristics. The interprocess communication and locality behavior of three of these applications can be determined statically, but they have different computation to communication ratios. For the other two applications, the locality and the interprocess communication are dependent on the input data and are not determinable statically. The applications thus span a diverse range of characteristics. The network topologies (full, cube, mesh) also have diverse connectivities. The observations from our study are summarized below:

On Network Abstractions

The separation of overheads provided by SPASM has helped us evaluate the use of L and g parameters of the LogP model for abstracting the network. In all the considered cases the latency overhead from the model and the target network closely agree. The pessimism in the model of assuming L to be the latency for the maximum size message on the network does not seem to have a significant impact on the accuracy of the latency overhead. Incidentally, we made a conscious decision in the cLogP simulation to abstract the specifics of the coherence protocol by ignoring the overheads associated with the coherence actions. The results show that the ensuing optimism does not impact the accuracy of the latency overhead either.

On the other hand, there is a disparity between the model and the target network for the contention overhead in many cases. The two sources of disparity are (a) the way g is computed, and (b) the way g is to be used as defined by the model. Since g is computed using only the bisection bandwidth of the network (as is suggested in [11]), it fails to capture any communication locality resulting from mapping the application on to a specific network topology. The ensuing pessimism in the observed contention overhead would increase with decreasing connectivity in the network as we have seen in the previous section. There is also a potential for the model to be optimistic with respect to the contention overhead when two distinct source-destination pairs share a common link. The second source of disparity leads purely to a pessimistic estimate of the contention overhead. The node architecture may have several ports that gives the potential for simultaneous network activity from a given node. However, the model definition precludes even simultaneous "sends" and "receives" from a given node.

As can be seen from our results, the pessimistic effects in computing and using g dominates the observed contention overheads. While it may be difficult to change the way g is computed within the confines of the LogP model, at least the way it is used should be modified to lessen the pessimism. For example, we conducted a simple experiment for FFT on the cube allowing for the g gap only between identical communication events (such as sends for instance). The resulting contention overhead was much closer to the real network.

The disparity in the contention prediction suggests that we need to incorporate application characteristics in computing g. For static applications like EP, IS and FFT, we may be able to use the computation and communication pattern in determining g. But for applications like CG and CHOLESKY, dynamism precludes such an analysis. On the other hand, since we are using these models in an execution driven simulation, we may be able get a better handle on calculating g. For instance, we may be able to maintain a history of the execution and use it to calculate g.

It would be interesting to investigate such issues in arriving at a better estimate.

On Locality Abstraction

As we expected, locality is an important factor in determining the performance of parallel programs and cannot be totally abstracted away for performance prediction or performance-conscious program development. But locality in parallel computation is much more difficult to model due to the additional degrees of freedom compared to sequential computation. Even for static applications, data alignment (several variables falling in the same cache block as observed in FFT) and temporal interleaving of memory accesses across processors, are two factors that make abstracting locality complex. In dynamic applications, this problem is exacerbated owing to factors such as dynamic scheduling and synchronization (implicit synchronization using condition variables and explicit synchronizers such as locks and barriers). It is thus difficult to abstract locality properties of parallel systems by a static theoretical or analytical model. Hence, in this study we explored the issue of using an abstraction for locality in a dynamic execution-driven simulation environment. In particular, we wanted to verify if a simple abstraction of a cache at each processing node that is maintained coherent without modeling the overheads for coherence maintenance would suffice to capture the locality properties of the system. Such an abstraction would try to capture the true communication characteristics of the application without modeling any hardware induced artifacts. Our results show that such an abstraction does indeed capture the locality of the system, closely modeling the communication in the target machine.

The network messages incurred in our abstraction for locality is representative of the minimum overhead that any invalidation-based cache coherence protocol may hope to achieve on a sequentially consistent memory model. We compared the performance of such an abstraction with a machine that incorporates a simple invalidation-based protocol. Even for this simple protocol, the results of the two agree closely over the chosen range of applications. The performance of a fancier cache coherence protocol that reduces network traffic on the target machine is expected to agree even closer with the chosen abstraction. This result suggests that cache coherence overhead is insignificant at least for the set of applications considered, and hence the associated coherence-related network activity can be abstracted out of the simulation. The applications that have been considered in this study employ the data parallel paradigm which is representative of a large class of scientific applications. In this paradigm, each processor works with a different portion of the data space, leading to lower coherence related traffic compared to applications where there is a more active sharing of the data space. It may be noted that Wood et al. [30] also present simulation results showing that the performance of a suite of applications is not very sensitive to different cache coherence protocols. Our results also suggest that for understanding the performance of parallel applications, it may be sufficient to use our abstraction for locality. However, further study with a wider suite of applications is required to validate these claims. Such a study can also help identify application characteristics that lend themselves to our chosen abstraction.

Importance of Separating Parallel System Overheads

The isolation and quantification of parallel system overheads provided by SPASM helped us address both of the above issues. For instance, even when total execution time curves were similar the latency and contention overhead curves helped us determine whether the model parameters were accurate in capturing the intended machine abstractions. One can experimentally determine the accuracy of the performance predicted by the LogP model as is done in [11] using the CM-5. However, this approach does not validate the individual parameters abstracted using the model. On the other hand, we were able to show that the g-parameter is pessimistic for calculating the contention overhead for several applications, and that the L-parameter can be optimistic or pessimistic depending on the application characteristics.

Speed of Simulation

Our main reason in studying the accuracy of abstractions is so that they may be used to speed up execution-driven simulations. Intuitively, one would think that the LogP machine described in this paper would execute the fastest since it is the most abstract of the three. But, our simulations of the LogP machine took a longer time to complete than those of the target machine. This is because such a model is very pessimistic due to ignoring data locality and the way it accounts for network contention. Hence, the simulation encountered considerably more events (non-local accesses which are cache 'hits' in the target and cLogP machines result in network accesses in the LogP machine) making it execute slower. On the other hand, the simulation of cLogP, which is less pessimistic, is indeed around 250-300% faster than the simulation of the target machine. This factor can represent a substantial saving given that execution-driven simulation of real applications can take an inordinate amount of time. For instance, the simulation of some of the data points for CHOLESKY take between 8-10 hours for the target machine. If we can reduce the pessimism in cLogP in modeling contention, we may be able to reduce the time for simulation even further.

8 Concluding Remarks

Abstractions of machine artifacts are useful in a number of settings. Execution-driven simulation is one such setting. This simulation technique is a popular vehicle for performance prediction because of its ability to capture the dynamic behavior of parallel systems. However, simulating every aspect of a parallel system in the context of real applications places a tremendous requirement on resource usage, both in terms of space and time. In this paper, we explored the use of abstractions in alleviating this problem. In particular, we explored the use of abstractions in modeling the interconnection network and locality properties of parallel systems in an execution-driven simulator. We used the recently proposed LogP model to abstract the interconnection network. We abstracted the locality in the system by modeling a coherent private cache without accounting for the overheads associated with coherence maintenance. We used five parallel scientific applications and hardware platforms with three different network topologies to evaluate the chosen abstractions. The results of our study show that the network latency overhead modeled by LogP is fairly accurate. On the other hand, the network contention estimate can become very pessimistic, especially in applications which exhibit communication locality. With regard to the data locality issue, the chosen simple abstraction for the cache does a good job in closely modeling the locality of the target machine over the chosen range of applications. The simulation speed of the model which incorporated these two abstractions was around 250-300% faster than the simulation of the target hardware, which can represent a sub-

stantial saving given that simulation of real parallel systems can take an inordinate amount of time.

References

[1] V. S. Adve and M. K. Vernon. Performance analysis of mesh interconnection networks with deterministic routing. *IEEE Transactions on Parallel and Distributed Systems*, 5(3):225–246, March 1994.

[2] A. Agarwal. Limits on Interconnection Network Performance. *IEEE Transactions on Parallel and Distributed Systems*, 2(4):398–412, October 1991.

[3] A. Agarwal et al. The MIT Alewife machine : A large scale Distributed-Memory Multiprocessor. In *Scalable shared memory multiprocessors*. Kluwer Academic Publishers, 1991.

[4] A. Aggarwal, A. K. Chandra, and M. Snir. On Communication Latency in PRAM Computations. In *Proceedings of the First Annual ACM Symposium on Parallel Algorithms and Architectures*, pages 11–21, 1989.

[5] H. Alt, T. Hagerup, K. Mehlhorn, and F. P. Preparata. Deterministic Simulation of Idealized Parallel Computers on More Realistic Ones. *SIAM Journal of Computing*, 16(5):808–835, 1987.

[6] T. E. Anderson. The Performance of Spin Lock Alternatives for Shared-Memory Multiprocessors. *IEEE Transactions on Parallel and Distributed Systems*, 1(1):6–16, January 1990.

[7] D. Bailey et al. The NAS Parallel Benchmarks. *International Journal of Supercomputer Applications*, 5(3):63–73, 1991.

[8] E. A. Brewer, C. N. Dellarocas, A. Colbrook, and W. E. Weihl. PROTEUS : A high-performance parallel-architecture simulator. Technical Report MIT-LCS-TR-516, Massachusetts Institute of Technology, Cambridge, MA 02139, September 1991.

[9] R. Cole and O. Zajicek. The APRAM: Incorporating Asynchrony into the PRAM Model. In *Proceedings of the First Annual ACM Symposium on Parallel Algorithms and Architectures*, pages 169–178, 1989.

[10] R. G. Covington, S. Madala, V. Mehta, J. R. Jump, and J. B. Sinclair. The Rice parallel processing testbed. In *Proceedings of the ACM SIGMETRICS 1988 Conference on Measurement and Modeling of Computer Systems*, pages 4–11, Santa Fe, NM, May 1988.

[11] D. Culler et al. LogP : Towards a realistic model of parallel computation. In *Proceedings of the 4th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, pages 1–12, May 1993.

[12] W. J. Dally. Performance analysis of k-ary n-cube interconnection networks. *IEEE Transactions on Computer Systems*, 39(6):775–785, June 1990.

[13] H. Davis, S. R. Goldschmidt, and J. L. Hennessy. Multiprocessor Simulation and Tracing Using Tango. In *Proceedings of the 1991 International Conference on Parallel Processing*, pages II 99–107, 1991.

[14] S. Fortune and J. Wyllie. Parallelism in random access machines. In *Proceedings of the 10th Annual Symposium on Theory of Computing*, pages 114–118, 1978.

[15] P. B. Gibbons. A More Practical PRAM Model. In *Proceedings of the First Annual ACM Symposium on Parallel Algorithms and Architectures*, pages 158–168, 1989.

[16] D. Lenoski, J. Laudon, K. Gharachorloo, W-D Weber, A. Gupta, J. Hennessy, M. Horowitz, and M. S. Lam. The Stanford DASH multiprocessor. *IEEE Computer*, 25(3):63–79, March 1992.

[17] P. Mehra, C. H. Schulbach, and J. C. Yan. A comparison of two model-based performance-prediction techniques for message-passing parallel programs. In *Proceedings of the ACM SIGMETRICS 1994 Conference on Measurement and Modeling of Computer Systems*, pages 181–190, May 1994.

[18] Microelectronics and Computer Technology Corporation, Austin, TX 78759. *CSIM User's Guide*, 1990.

[19] J. H. Patel. Analysis of multiprocessors with private cache memories. *IEEE Transactions on Computer Systems*, 31(4):296–304, April 1982.

[20] S. K. Reinhardt et al. The Wisconsin Wind Tunnel : Virtual prototyping of parallel computers. In *Proceedings of the ACM SIGMETRICS 1993 Conference on Measurement and Modeling of Computer Systems*, pages 48–60, Santa Clara, CA, May 1993.

[21] E. Rothberg, J. P. Singh, and A. Gupta. Working sets, cache sizes and node granularity issues for large-scale multiprocessors. In *Proceedings of the 20th Annual International Symposium on Computer Architecture*, pages 14–25, May 1993.

[22] J. P. Singh, E. Rothberg, and A. Gupta. Modeling communication in parallel algorithms: A fruitful interaction between theory and systems? In *Proceedings of the Sixth Annual ACM Symposium on Parallel Algorithms and Architectures*, 1994.

[23] J. P. Singh, W-D. Weber, and A. Gupta. SPLASH: Stanford Parallel Applications for Shared-Memory. Technical Report CSL-TR-91-469, Computer Systems Laboratory, Stanford University, 1991.

[24] A. Sivasubramaniam, A. Singla, U. Ramachandran, and H. Venkateswaran. Abstracting network characteristics and locality properties of parallel systems. Technical Report GIT-CC-93/63, College of Computing, Georgia Institute of Technology, October 1993.

[25] A. Sivasubramaniam, A. Singla, U. Ramachandran, and H. Venkateswaran. An Approach to Scalability Study of Shared Memory Parallel Systems. In *Proceedings of the ACM SIGMETRICS 1994 Conference on Measurement and Modeling of Computer Systems*, pages 171–180, May 1994.

[26] A. Sivasubramaniam, A. Singla, U. Ramachandran, and H. Venkateswaran. On characterizing bandwidth requirements of parallel applications. Technical Report GIT-CC-94/31, College of Computing, Georgia Institute of Technology, July 1994.

[27] A. Sivasubramaniam, A. Singla, U. Ramachandran, and H. Venkateswaran. A Simulation-based Scalability Study of Parallel Systems. *Journal of Parallel and Distributed Computing*, 22(3):411–426, September 1994.

[28] L. G. Valiant. A Bridging Model for Parallel Computation. *Communications of the ACM*, 33(8):103–111, August 1990.

[29] H. Wabnig and G. Haring. PAPS - The Parallel Program Performance Prediction Toolset. In *Proceedings of the 7th International Conference on Modeling Techniques and Tools for Computer Performance Evaluation*, Vienna, Austria, May 1994.

[30] D. A. Wood et al. Mechanisms for cooperative shared memory. In *Proceedings of the 20th Annual International Symposium on Computer Architecture*, pages 156–167, May 1993.

[31] J. C. Wyllie. *The Complexity of Parallel Computations*. PhD thesis, Department of Computer Science, Cornell University, Ithaca, NY, 1979.

To appear in *Proceedings of the First International Workshop on Parallel Processing*, Bangalore, India, December 1994.

Issues in Understanding the Scalability of Parallel Systems*

Umakishore Ramachandran *H. Venkateswaran* *Anand Sivasubramaniam* *Aman Singla*

College of Computing
Georgia Institute of Technology
Atlanta, GA 30332-0280.
{*rama, venkat, anand, aman*}*@cc.gatech.edu*

(Extended Abstract)

1 Introduction

Scalability is a term frequently used to qualify the match between an algorithm and architecture in a parallel system (an algorithm-architecture combination). Evaluating the scalability of a parallel system has widespread applicability. The results from such an evaluation may be used to: select the best architecture platform for an application domain, predict the performance of an application on a larger configuration of an existing architecture, identify application and architectural bottlenecks in a parallel system to suggest application restructuring and architectural enhancements, and glean insight on the interaction between an application and an architecture to understand the scalability of other application-architecture pairs. But evaluating and predicting the scalability of parallel systems poses several problems due to the complex interaction between application characteristics and architectural features. In this paper, we propose an approach for evaluating the scalability of parallel systems and develop a framework for studying the inter-play between applications and architectures. Using this framework, we study the scalability of five parallel scientific applications on shared memory platforms with three different network topologies. We illustrate the power of this framework in addressing two related issues. First, we use it to evaluate abstractions of parallel systems that have been proposed for modeling parallel system behavior. Second, we show its important use in synthesizing architectural requirements from an application perspective.

Since real-life applications set the standards for computing, our approach uses such applications for studying the scalability of parallel systems. We call such an application-driven approach a *top-down approach to scalability study*. The main thrust of this approach is to identify important algorithmic and architectural artifacts that impact the performance of a parallel system, understand the interaction between them, quantify the impact of these artifacts on the execution time of an application, and use these quantifications in studying the scalability of the system. We associate an *overhead function* with each algorithmic and architectural artifact that impedes the performance of a parallel system. We isolate and quantify the algorithmic overheads such as serial fraction and work-imbalance from the overall execution time of an application. We also isolate other overheads such as network *latency* (the actual hardware transmission time in the network) and network *contention* (the amount of time spent waiting for a resource to become free in the network) arising from the interaction of the algorithm with the underlying hardware. Our approach uses a combination of experimentation, simulation and analytical techniques in quantifying these overheads.

*This work has been funded in part by NSF grants MIPS-9058430 and MIPS-9200005, and an equipment grant from DEC.

Traditional performance metrics such as speedup [1], scaled speedup [8], sizeup [23], experimentally determined serial fraction [9], and isoefficiency function [10], are useful for tracking performance trends, but they do not provide adequate information needed to understand the reason why an application does not scale well on an architecture. The overhead functions that we identify, separate, and quantify, help us overcome this inadequacy. The growth of overhead functions as a function of system parameters can provide key insights on the scalability of a parallel system by suggesting application restructuring, as well as architectural enhancements. Crovella and LeBlanc [6] follow a similar approach towards quantifying cycles that are lost due to different overheads in a parallel system using experimentation. Our approach uses simulation to isolate parallel system overheads. The importance of simulation in capturing the dynamics of parallel system interactions has been addressed in [17, 14, 13, 4, 5].

This work is part of an on-going project which aims at understanding the significant issues in the design of scalable parallel systems using the above-mentioned top-down approach. In our earlier work, we studied issues such as task granularity, data distribution, scheduling, and synchronization, by implementing frequently used parallel algorithms on shared memory [18] and message-passing [16] platforms. In [21], we illustrate the top-down approach for the scalability study of message-passing systems. In [20], we conduct a similar study for shared memory systems. The utility of the framework in evaluating machine abstractions and synthesizing network requirements are presented in [19] and [22] respectively.

The top-down approach and the overhead functions are elaborated in Section 2. The different ways of implementing this approach and details of a simulation platform, SPASM (Simulator for Parallel Architectural Scalability Measurements), which quantifies these overhead functions, are also discussed in this section. Using a set of five parallel applications and three hardware platforms, we summarize the use of our framework in studying the scalability of parallel systems (section 3.1), evaluating the validity of abstractions (section 3.2) and synthesizing network requirements from an application perspective (section 3.3). Concluding remarks are presented in Section 4.

2 Top-Down Approach

Adhering to the RISC ideology in the evolution of sequential architectures, we would like to use *real world applications* in the performance evaluation of parallel machines. However, applications normally tend to contain large volumes of code that are not easily portable and a level of detail that is not very familiar to someone outside that application domain. Hence, computer scientists have traditionally used parallel algorithms that capture

interesting computation phases of applications for benchmarking their machines. Such abstractions of real applications which capture the main phases of the computation are called *kernels*. One can go even lower than kernels by abstracting the main *loops* in the computation (like the Lawrence Livermore loops [11]) and evaluating their performance. As one goes lower, the outcome of the evaluation becomes less realistic. Even though an application may be abstracted by the kernels inside it, the sum of the times spent in the underlying kernels may not necessarily yield the time taken by the application. There is usually a cost involved in moving from one kernel to another such as the data movements and rearrangements in an application that are not part of the kernels that it is comprised of. For instance, an efficient implementation of a kernel may need to have the input data organized in a certain fashion which may not necessarily be the format of the output from the preceding kernel in the application. Despite its limitations, we believe that the scalability of an application with respect to an architecture can be captured by studying its kernels, since they represent the computationally intensive phases of an application. Hence, we have used kernels in the subsequent studies.

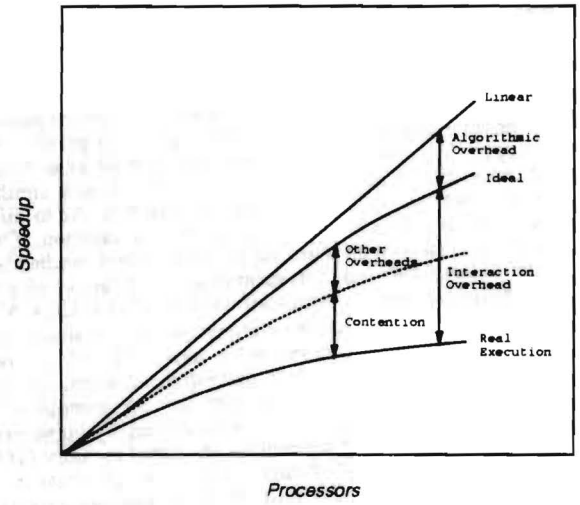

Figure 1: Top-down Approach to Scalability Study

Parallel system overheads (see Figure 1) may be broadly classified into a purely algorithmic component (*algorithmic overhead*), and a component arising from the interaction of the algorithm and the architecture (*interaction overhead*). Algorithmic overhead is due to the inherent *serial* part [1] and the *work-imbalance* in the algorithm, and is independent of architectural characteristics. Isolating these two components of the algorithmic overhead can help in re-structuring the algorithm. The algorithmic overhead is quantified by computing the time taken for execution of a given parallel program on an ideal machine such as the PRAM [24] and measuring its deviation from a linear speedup curve. A real execution could deviate significantly from the ideal execution due to overheads such as latency, contention, synchronization, scheduling and cache effects. These overheads are lumped together as the interaction overhead. In an architecture with no contention overhead, the communication pattern of the application would dictate the latency overhead incurred by it. Thus the performance of an application (on an architecture devoid of network contention) may lie between the ideal curve and the real execution curve (see Figure 1). To fully understand the scalability of a parallel system, it is important to further isolate the influence of each component of the interaction overhead on the overall performance. For this purpose, we have introduced the notion of *overhead functions* that allows separation and quantification of these bottlenecks. An overhead function quantifies the growth of a specific overhead in the parallel system as a function of system parameters. *Constant problem size* (where the problem size remains unchanged as the number of processors is increased), *memory constrained* (where the problem size is scaled up linearly with the number of processors), and *time constrained* (where the problem size is scaled up to keep the execution time constant with increasing number of processors) are three well-accepted scaling models used in the study of parallel systems. Overhead functions can be used to study the growth of system overheads for any of these scaling strategies.

The key elements of our top-down approach for studying the scalability of parallel systems are:

- experiment with real world applications
- identify parallel kernels that occur in these applications
- study the interaction of these kernels with architectural features to separate and quantify the overheads in the parallel system
- use these overheads for predicting the scalability of parallel systems.

2.1 Implementing the Top-Down Approach

	Statistics	Accuracy	Space/Time
Experimentation	(Low)	High	Low
Analytical Methods	High	(Low)	Low
Simulation	High	High	(High)

Table 1: Comparing the Implementation Approaches

Scalability study of parallel systems is complex due to the several degrees of freedom that they exhibit. Experimentation, simulation, and analytical models are three techniques that have been commonly used in such studies. But it is well-known that each has its relative merits and de-merits [17]. Table 1 classifies these techniques in terms of the amount of statistics that can be obtained, the accuracy of these statistics, and the effort (space and time) expended in each evaluation technique. The amount of statistics that can be obtained by experimenting with applications on actual machines is largely limited by the monitoring support provided by the underlying hardware. Further, the underlying hardware is fixed, making it difficult to study the effect of changing individual architectural parameters on the performance. Instrumenting the code may also become intrusive affecting the accuracy of the results. Analytical models can provide a wide range of statistical information at a moderately low cost. But, it is not clear that such models can realistically capture the complex and dynamic interactions between applications and architectures. Finally, simulation has the advantage of providing quite accurate results over a large set of statistics. But it does require considerable computational resources in terms of space and time to simulate large systems. We use a combination of all three for implementing the top-down approach as shown in Figure 2. Experimentation is used in conjunction with simulation to understand the performance of real applications on real architectures, and to identify the interesting kernels that occur in these applications for subsequent use in the simulation studies. We use the datapoints obtained from simulation to develop, validate and refine analytical models and use these to predict the scalability of larger systems. Refined models

of parallel system artifacts thus derived may also be used to abstract features in the application and simulated hardware to speed up the simulation.

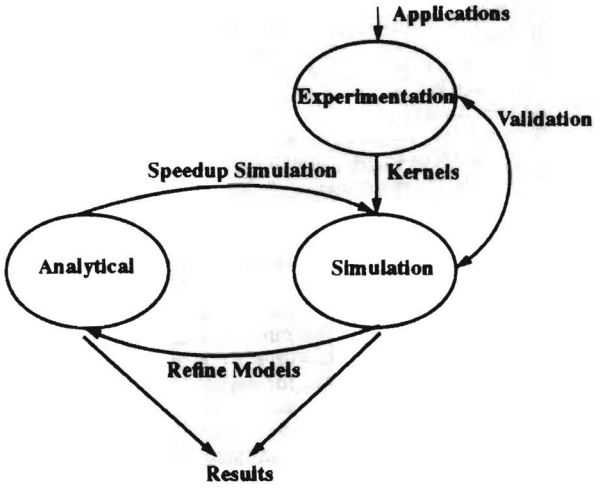

Figure 2: Framework

At the heart of our framework lies a simulation platform called SPASM, that provides an elegant set of mechanisms for quantifying the different overheads. Details of this simulation platform in the context of simulating shared memory platforms are presented in the next subsection. The reader is referred to [21] for the capabilities of SPASM in simulating message-passing platforms.

2.2 SPASM

SPASM is an execution-driven simulator written in CSIM [12]. As with other recent simulators [4, 5, 13], the bulk of the instructions in the parallel program is executed at the speed of the native processor (SPARC in this study) and only the instructions (such as LOADS and STORES) that may potentially involve a network access are simulated. The input to the simulator are parallel applications written in C. These programs are pre-processed (to label shared memory accesses), the compiled assembly code is augmented with cycle counting instructions, and the assembled binary is linked with the simulator code. The system parameters that can be specified to SPASM are: the *number of processors* (p), the *clock speed* of the processor, the *network topology*, the *hardware bandwidth* of the links in the network, and the *switching delays*.

2.2.1 Metrics

SPASM provides a wide range of statistical information about the execution of the program. It gives the *total time* (simulated time) which is the maximum of the running times of the individual parallel processors. This is the time that would be taken by an execution of the parallel program on the target parallel machine. *Speedup* using p processors is measured as the ratio of the total time on 1 processor to the total time on p processors.

Ideal time is the total time taken by a parallel program to execute on an ideal machine such as the PRAM. It includes the algorithmic overhead but does not include the interaction overhead. SPASM simulates an ideal machine to provide this metric. As we mentioned in Section 2, the difference between the linear time and the ideal time gives the algorithmic overhead.

SPASM also quantifies the different components of the interaction overhead. Accesses to variables in a shared memory system may involve the network, and the physical limitations of the network tend to contribute to overheads in the execution. These overheads may be broadly classified as latency and contention, and we associate an overhead function with each. The *Latency Overhead Function* is thus defined as the total amount of time spent by a processor waiting for messages due to the transmission time on the links and the switching overhead in the network assuming that the messages did not have to contend for any link. Likewise, the *Contention Overhead Function* is the total amount of time incurred by a processor due to the time spent waiting for links to become free by the messages. SPASM quantifies both the latency overhead function as well as the contention overhead function seen by a processor. This is done by time-stamping messages when they are sent. At the time a message is received, the time that the message would have taken in a contention free environment is charged to the latency overhead function while the rest of the time is accounted for in the contention overhead function. Though not relevant to this study, it is worthwhile to mention that SPASM provides the latency and contention incurred by a message as well as the latency and contention that a processor may choose to see. Even though a message may incur a certain latency and contention, a processor may choose to hide all or part of it by overlapping computation with communication. Such a scenario may arise with a non-blocking message operation on a message-passing machine or with a prefetch operation on a shared memory machine. But for the rest of this paper (since we deal with blocking load/store shared memory operations), we assume that a processor sees all of the network latency and contention.

Shared memory systems normally provide some synchronization support that is as simple as an atomic read-modify-write operation, or may provide special hardware for more complicated operations like barriers and queue-based locks. While the latter may save execution time for complicated synchronization operations, the former is more flexible for implementing a variety of such operations. For reasons of generality, we assume that only the test&set operation is supported by shared memory systems. We also assume that the memory module (at which the operation is performed), is intelligent enough to perform the necessary operation in unit time. With such an assumption, the only network overhead due to the synchronization operation (test&set) is a roundtrip message, and the overheads for such a message are accounted for in the latency and contention overhead functions described earlier. The waiting time incurred by a processor during synchronization operations is accounted for in the CPU time which would manifest itself as an algorithmic overhead.

SPASM also provides statistical information about the network. It gives the utilization of each link in the network and the average queue lengths of messages at any particular link. This information can be useful in identifying network bottlenecks and comparing relative merits of different networks and their capabilities.

It is often useful to have the above metrics for different modes of execution of the algorithm. Such a breakup would help identify bottlenecks in the program, and also help estimate the potential gain in performance that may be possible through a specific hardware or software enhancement. SPASM provides statistics grouped together for system-defined as well as for user-defined modes of execution. The statistics are collected by SPASM for each processor individually for these modes. The results presented in this paper are for a representative processor. The system-defined modes are:

- BARRIER: Mode corresponding to a barrier synchronization operation.
- MUTEX: Even though the simulated hardware provides only a test&set operation, mutual exclusion *lock* (implemented using test-test&set [2]) is available as a library function in SPASM. A program enters this mode during lock operations. With this mechanism, we can separate the overheads due to the synchronization operations from the rest of the program execution.

- PGM_SYNC: Parallel programs may use Signal-Wait semantics for pairwise synchronization. A lock is unnecessary for the Signal variable since only 1 processor writes into it and the other reads from it. This mode is used to differentiate such accesses from normal load/store accesses.

- NORMAL: A program is in the NORMAL mode if it is not in any of the other modes. An application programmer may further define sub-modes if necessary.

The *total time* for a given application is the sum of the *execution times* for each of the above defined modes. The *execution time* for each program mode is the sum of the *computation time*, the *latency overhead* and the *contention overhead* observed in the mode. Computation time in the NORMAL mode is the actual time spent in local computation in an application. The sum of latency and contention overheads in the NORMAL mode is the actual time incurred for ordinary data accesses. For the BARRIER and PGM_SYNC modes, the computation time is the wait time incurred by a processor in synchronizing with other processors that results from the algorithmic work imbalance. The computation time in the MUTEX mode is the time spent in waiting for a lock and represents the serial part in an application arising due to critical sections. For the BARRIER and MUTEX modes, the computation time also includes the cost of implementing the synchronization primitive and other residual effects due to latency and contention for prior accesses. In all three synchronization modes, the latency and contention overheads together represent the actual time incurred in accessing synchronization variables. The metrics identified by SPASM thus quantify the interesting components of the algorithmic and interaction overheads.

3 Uses of the Framework

In illustrating the use of our framework, we use a diverse range of applications and hardware platforms. Three of the applications (EP, IS and CG) are from the NAS parallel benchmark suite [3]; CHOLESKY is from the SPLASH benchmark suite [15]; and FFT is the well-known Fast Fourier Transform algorithm. EP and FFT are well-structured applications with regular communication patterns determinable at compile-time, with the difference that EP has a higher computation to communication ratio. IS also has a regular communication pattern, but in addition it uses locks for mutual exclusion during the execution. CG and CHOLESKY are different from the other applications in that their communication patterns are not regular (both use sparse matrices) and cannot be determined at compile time. While a certain number of rows of the matrix in CG is assigned to a processor at compile time (static scheduling), CHOLESKY uses a dynamically maintained queue of runnable tasks. For the underlying hardware, we use shared memory platforms with three different network topologies: a fully connected network, a binary hypercube and a 2-D mesh.

The framework described in Section 2 has widespread applicability. It can be used in performance debugging to identify application and architectural bottlenecks, suggesting application restructuring and architectural enhancements. It can be used to predict the performance of the application over a range of system parameters. The scalability of the above applications on the chosen hardware platforms is summarized in Section 3.1. The framework can be used to develop new analytical models, and to validate and refine existing analytical/theoretical models for parallel systems. In Section 3.2, we illustrate the use of the framework in validating models chosen for abstracting the network characteristics and locality properties of parallel systems. The framework can also help in synthesizing architectural requirements from an application viewpoint, which is very important for building well-balanced machines. In particular, the use of the framework in synthesizing network requirements for the chosen applications in presented in section 3.3.

3.1 Scalability Study of Parallel Systems

In [20] we illustrated the use of the framework in studying the scalability of the five parallel applications on the three simulated platforms. We separated and quantified the different overheads in the parallel system, and developed models to capture the growth of overheads with system parameters. The resulting overhead functions helped us identify and quantify the algorithmic and architectural bottlenecks in the parallel systems. The results from the study are summarized below.

EP displays a sufficiently high computation to communication ratio, with the computation time dominating over the latency and contention overheads in the network. Further, the algorithmic overheads in this application are negligible, resulting in a scalable execution with increasing processors across all hardware platforms.

Parallelization of IS increases the amount of work to be done for a given problem size. This inherent algorithmic overhead causes a deviation of the ideal curve from the linear curve, making the application unscalable for small problem sizes. On a fully connected network, the contention overhead is negligible and the latency converges to a constant with a sufficiently large number of processors. Thus, the scalability of this kernel on the fully connected network is expected to closely follow the ideal curve. For the hypercube the contention overhead grows logarithmically with the number of processors, while for the mesh this growth is linear, thus worsening the scalability of this application on these two platforms.

In FFT, the algorithmic overheads are marginal and the latency overhead decreases with increasing number of processors. Thus the contention overhead is the only artifact that can cause deviation from linear behavior. The communication in FFT is limited to a single phase where every processor communicates with every other processor. But these communication steps are skewed, and the network contention begins to show only on the mesh network where it grows linearly. FFT is thus scalable for the fully-connected and cube platforms. For the mesh platform, it would take 200 processors before the contention overhead starts dominating for a 64K problem size. Increasing the problem size improves its scalability on all three platforms.

For CG, the latency overhead decreases with increasing number of processors while the contention overhead is more pronounced. The contention overhead is negligible for the fully-connected network, grows linearly for the cube and the mesh, with a larger coefficient for the mesh compared to the cube. CG is thus scalable for the fully-connected network and becomes less scalable for networks with lower connectivity like the cube and the mesh.

CHOLESKY is not very scalable for the chosen problem size due to the inherent algorithmic overheads. Of the interaction overheads, latency decreases with increasing number of processors, making the contention component dictate the scalability of this application. The contention on the fully-connected and cube networks is negligible thus projecting speedup curves that closely follow the ideal speedup curve for these platforms. On the other hand, the contention grows logarithmically on the mesh making this platform less scalable.

Isolation and separation of the different overheads thus helped us identify and quantify application and architectural bottlenecks. Identifying such bottlenecks can suggest application restructuring and architectural enhancements. For instance, an initial implementation of IS exhibited a substantial contention overhead. An examination of the overhead functions over the course of the execution helped us restructure the implementation to reduce this overhead.

3.2 Validating Abstractions of Parallel Systems

Abstracting features of parallel systems is a technique often employed in performance analysis and algorithm development. For instance, abstracting parallel machines by theoretical models like

the PRAM [24] has facilitated algorithm development and analysis. Such models try to hide hardware details from the programmer, providing a simplified view of the machine. Similarly, analytical models used in performance evaluation abstract complex system interactions with simple mathematical functions, parameterized by a limited number of degrees of freedom that are tractable. Abstractions are also useful in execution-driven simulators where details of the hardware and the application can be captured by abstract models in order to ease the demands on resource (time and space) usage in simulating large parallel systems. Some simulators [20, 4, 5, 13] already abstract details of instruction-set simulation, since such a detailed simulation is not likely to contribute significantly to the performance analysis of parallel systems.

An important question that needs to be addressed in using abstractions is their validity. Our framework serves as a convenient vehicle for evaluating the accuracy of these abstractions using real applications. In [19], we illustrate the use of the framework to evaluate the validity and use of abstractions in simulating the interconnection network and locality properties of parallel systems. An outline of the evaluation strategy and results are presented below.

For abstracting the interconnection network, we use the recently proposed *LogP* [7] model that incorporates the two defining characteristics of a network, namely, latency and contention. For abstracting the locality properties of a parallel system, we model a private cache at each processing node in the system to capture data locality. Shared memory machines with private caches usually employ a protocol to maintain coherence. With a diverse range of cache coherence protocols, it would become very specific if our abstraction were to model any particular protocol. Further, memory references (locality) are largely dictated by application characteristics and are relatively independent of cache coherence protocols. Hence, instead of modeling any particular protocol, we choose to maintain the caches coherent in our abstraction but do not model the overheads associated with maintaining the coherence. Such an abstraction would represent an ideal coherent cache that captures the true inherent locality in an application. Furthermore, if our abstraction closely models the behavior of a machine with a simple cache coherent protocol, then it would even more closely model the behavior of a machine with a fancier cache coherence protocol.

We use our simulation framework for evaluating these abstractions. We compare the results from simulating the five applications on a machine incorporating these abstractions with the results from an exact simulation of the actual hardware. Our results show that the latency overhead modeled by LogP is fairly accurate. On the other hand, the contention overhead modeled by LogP can become pessimistic for some applications since the model does not capture communication locality. The pessimism gets amplified as we move to networks with lower connectivity. With regard to the data locality question, results show that our ideal cache, which does not model any coherence protocol overheads, is a good abstraction for capturing locality over the chosen range of applications.

Apart from evaluating these abstractions in the context of real applications, the isolation and quantification of parallel system overheads helps us validate the individual parameters used in each abstraction. For instance, even when total execution time curves were similar, the latency and contention overheads helped us determine whether the LogP parameters were accurate in capturing the intended machine abstractions. The simulation of the system which incorporates these two abstractions is around 250-300% faster than the simulation of the actual machine. This factor can represent a substantial saving given that execution-driven simulation of real applications can take an inordinate amount of time. Using a similar approach, one may also use this framework to refine existing models (like reducing the pessimism in LogP in modeling contention), or even develop new models for accurately capturing parallel system behavior.

3.3 Synthesizing Network Requirements

For building a general-purpose parallel machine, it is essential to identify and quantify the architectural requirements necessary to assure good performance over a wide range of applications. Such a synthesis of requirements from an application view-point can help us make cost vs. performance trade-offs in important architectural design decisions. Our framework provides a convenient platform to study the impact of hardware parameters on application performance and use the results to project architectural requirements. We conducted such a study in [22] towards synthesizing the network requirements of the applications mentioned earlier, and the experimental strategy along with interesting results from our study are summarized here.

To quantify link bandwidth requirements for a particular network topology, we simulate the execution of the applications on such a topology and vary the bandwidth of the links in the network. As the bandwidth is increased, the network overheads (latency and contention) decrease, yielding a performance that is close to the ideal execution. From these results, we arrive at link bandwidths that are needed to limit network overheads (latency and contention) to an acceptable level of the overall execution time. We also study the impact of the number of processors, the CPU clock speed and the application problem size on bandwidth requirements. Computation to communication ratio tends to decrease when the number of processors or the CPU clock speed is increased, making the network requirements more stringent. An increase in problem size improves the computation to communication ratio, lowering the bandwidth needed to maintain an acceptable efficiency. Using regression analysis and analytical techniques, we extrapolate requirements for systems built with larger number of processors.

The results from the study suggest that existing link bandwidth of 200-300 MBytes/sec available on machines like Intel Paragon and Cray T3D can easily sustain the requirements of two applications (EP and FFT) even on high-speed processors of the future. For the other three, one may be able to maintain network overheads at an acceptable level if the problem size is increased commensurate with the processing speed.

The separation of the overheads plays an important role in synthesizing the communication requirements of applications. For instance, an application may have an algorithmic deficiency due to either a large serial part or due to work-imbalance, in which case 100% efficiency is impossible regardless of other architectural parameters. The separation of overheads enables us to quantify bandwidth requirements as a function of acceptable network overheads (latency and contention). The framework may also be used for synthesizing requirements of other architectural features such as synchronization primitives and locality capabilities from an application perspective.

4 Concluding Remarks

In this paper, we presented a novel approach for studying the scalability of parallel systems using real-world applications. Our approach uses a combination of experimentation, analytical modeling and simulation towards identifying, isolating and quantifying the different overheads in a parallel system that limit its scalability. We described an execution-driven simulation platform that can separate the interesting components of the algorithmic and interaction overheads from the overall execution time. Using a set of five parallel applications and three hardware platforms, we illustrated the use of our approach and simulation framework in 1) studying the scalability of these applications on the chosen hardware platforms; 2) evaluating the validity of parallel system abstractions; and 3) synthesizing network requirements from an application perspective.

References

[1] G. M. Amdahl. Validity of the Single Processor Approach to achieving Large Scale Computing Capabilities. In *Proceedings of the AFIPS Spring Joint Computer Conference*, pages 483–485, April 1967.

[2] T. E. Anderson. The Performance of Spin Lock Alternatives for Shared-Memory Multiprocessors. *IEEE Transactions on Parallel and Distributed Systems*, 1(1):6–16, January 1990.

[3] D. Bailey et al. The NAS Parallel Benchmarks. *International Journal of Supercomputer Applications*, 5(3):63–73, 1991.

[4] E. A. Brewer, C. N. Dellarocas, A. Colbrook, and W. E. Weihl. PROTEUS : A high-performance parallel-architecture simulator. Technical Report MIT-LCS-TR-516, Massachusetts Institute of Technology, Cambridge, MA 02139, September 1991.

[5] R. G. Covington, S. Madala, V. Mehta, J. R. Jump, and J. B. Sinclair. The Rice parallel processing testbed. In *Proceedings of the ACM SIGMETRICS 1988 Conference on Measurement and Modeling of Computer Systems*, pages 4–11, Santa Fe, NM, May 1988.

[6] M. E. Crovella and T. J. LeBlanc. Parallel Performance Prediction Using Lost Cycles Analysis. In *Proceedings of Supercomputing '94*, November 1994.

[7] D. Culler et al. LogP : Towards a realistic model of parallel computation. In *Proceedings of the 4th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, pages 1–12, May 1993.

[8] J. L. Gustafson, G. R. Montry, and R. E. Benner. Development of Parallel Methods for a 1024-node Hypercube. *SIAM Journal on Scientific and Statistical Computing*, 9(4):609–638, 1988.

[9] A. H. Karp and H. P. Flatt. Measuring Parallel processor Performance. *Communications of the ACM*, 33(5):539–543, May 1990.

[10] V. Kumar and V. N. Rao. Parallel Depth-First Search. *International Journal of Parallel Programming*, 16(6):501–519, 1987.

[11] F. H. McMahon. The Livermore Fortran Kernels : A Computer Test of the Numerical Performance Range. Technical Report UCRL-53745, Lawrence Livermore National Laboratory, Livermore, CA, December 1986.

[12] Microelectronics and Computer Technology Corporation, Austin, TX 78759. *CSIM User's Guide*, 1990.

[13] S. K. Reinhardt et al. The Wisconsin Wind Tunnel: Virtual prototyping of parallel computers. In *Proceedings of the ACM SIGMETRICS 1993 Conference on Measurement and Modeling of Computer Systems*, pages 48–60, Santa Clara, CA, May 1993.

[14] J. P. Singh, E. Rothberg, and A. Gupta. Modeling communication in parallel algorithms: A fruitful interaction between theory and systems? In *Proceedings of the Sixth Annual ACM Symposium on Parallel Algorithms and Architectures*, 1994.

[15] J. P. Singh, W-D. Weber, and A. Gupta. SPLASH: Stanford Parallel Applications for Shared-Memory. Technical Report CSL-TR-91-469, Computer Systems Laboratory, Stanford University, 1991.

[16] A. Sivasubramaniam, U. Ramachandran, and H. Venkateswaran. Message-Passing: Computational Model, Programming Paradigm, and Experimental Studies. Technical Report GIT-CC-91/11, College of Computing, Georgia Institute of Technology, February 1991.

[17] A. Sivasubramaniam, U. Ramachandran, and H. Venkateswaran. A comparative evaluation of techniques for studying parallel system performance. Technical Report GIT-CC-94/38, College of Computing, Georgia Institute of Technology, September 1994.

[18] A. Sivasubramaniam, G. Shah, J. Lee, U. Ramachandran, and H. Venkateswaran. Experimental Evaluation of Algorithmic Performance on Two Shared Memory Multiprocessors. In Norihisa Suzuki, editor, *Shared Memory Multiprocessing*, pages 81–107. MIT Press, 1992.

[19] A. Sivasubramaniam, A. Singla, U. Ramachandran, and H. Venkateswaran. Abstracting network characteristics and locality properties of parallel systems. Technical Report GIT-CC-93/63, College of Computing, Georgia Institute of Technology, October 1993.

[20] A. Sivasubramaniam, A. Singla, U. Ramachandran, and H. Venkateswaran. An Approach to Scalability Study of Shared Memory Parallel Systems. In *Proceedings of the ACM SIGMETRICS 1994 Conference on Measurement and Modeling of Computer Systems*, pages 171–180, May 1994.

[21] A. Sivasubramaniam, A. Singla, U. Ramachandran, and H. Venkateswaran. A Simulation-based Scalability Study of Parallel Systems. *Journal of Parallel and Distributed Computing*, 1994. To appear.

[22] A. Sivasubramaniam, A. Singla, U. Ramachandran, and H. Venkateswaran. Synthesizing network requirements using parallel scientific applications. Technical Report GIT-CC-94/31, College of Computing, Georgia Institute of Technology, July 1994.

[23] X-H. Sun and J. L. Gustafson. Towards a better Parallel Performance Metric. *Parallel Computing*, 17:1093–1109, 1991.

[24] J. C. Wyllie. *The Complexity of Parallel Computations*. PhD thesis, Department of Computer Science, Cornell University, Ithaca, NY, 1979.

Timepatch: A Novel Technique for the Parallel Simulation of Multiprocessor Caches*

Gautam Shah
Umakishore Ramachandran
Richard Fujimoto

e-mail: {gautam, rama, fujimoto}@cc.gatech.edu

Technical Report GIT-CC-94-52

College of Computing
Georgia Institute of Technology
Atlanta, GA 30332-0280 USA

(For limited distribution, submitted for publication)

Abstract

We present a new technique for the parallel simulation of cache coherent shared memory multiprocessors. Our technique is based on the fact that the functional correctness of the simulation can be decoupled from its timing correctness. Thus in our simulations we can exploit as much parallelism as is available in the application without being constrained by conservative scheduling mechanisms that might limit the available parallelism in order to guarantee the timing correctness of the simulation. Further, application specific details (which can be gleaned from the compiler) such as data layout in the caches of the target architecture can be exploited to reduce the overhead of the simulation. The simulation correctness is guaranteed by patching the performance related timing information at specific points in the program (commensurate with the programming model). There are two principal advantages to this technique: being able to simulate larger parallel systems (both problem size and number of processors) than is feasible to simulate sequentially; and being able to speed up the simulation compared to a sequential simulator. For proof of concept, we have implemented this technique for an execution-driven parallel simulator on the KSR-2, a cache-coherent shared memory machine, for a target architecture that uses an invalidation-based protocol. We validate the performance statistics gathered from this simulator (using traces) by comparing it against a sequential simulator. We show that the method is both viable and promises to offer significant speedups with the number of processors. We provide a detailed performance study of our technique using some benchmark application programs.

Key Words:
Parallel simulation, performance evaluation, cache consistency, execution-driven simulation, shared memory multiprocessors, performance debugging.

*This work has been funded in part by NSF PYI award MIPS-9058430 and a matching equipment grant from DEC.

1 Introduction

Shared memory multiprocessors usually have private caches associated with each processor. There are many parameters to be tuned with respect to the cache design such as the cache size, the line size, associativity, the replacement policy, and the protocol used for cache coherence. Thus cache simulations play a very important role in the design cycle of building shared memory multiprocessors by aiding the choice of appropriate parameter values for a specific cache protocol and estimating the performance of the system. Various simulation techniques including trace-driven [EK88, ASHH88], and execution-driven [Fuj83, CMM+88, DGH91] methods have been used for this purpose. Most of the known approaches to cache simulation are sequential. Such simulations impose a heavy burden on system resources both in terms of space and time. The elapsed time for the simulation is particularly limiting on the size of the system that can be simulated with realistic workloads. Given the availability of commercial multiprocessors it is attractive to consider their use in reducing the elapsed time for the simulation by parallelizing the simulation itself. The expected benefit is in being able to simulate larger parallel systems (both number of processors and problem size) than can be feasibly simulated sequentially (due to space and time constraints), and the potential for speeding up the simulation compared to a sequential simulator.

Parallel simulation techniques are viable if they result in speedups as more number of processors are employed in the simulation. Typically these techniques fall into two categories - *conservative* and *optimistic*. In conservative techniques two parallel units of work can be scheduled at the same time if and only if one is guaranteed not to affect the execution of the other [Fuj90, CM79]. From the point of view of simulating cache-coherent multiprocessors, such a restriction invariably inhibits the simulation from being able to exploit the available parallelism in the application. On the other hand, optimistic scheduling techniques such as Time Warp [Jef85] have not been used for simulating shared memory systems because a naive application of this technique could result in considerable state saving overhead that may dominate the execution.

We develop a new technique, called *timepatch*, that exploits the available parallelism in the application for driving the parallel simulation of caches for shared memory multiprocessors. The method is based on using application specific knowledge to yield a mechanism that is conservative with respect to generating a correct sequence of instruction execution (commensurate with the programming model), but is optimistic with respect to the timing information. Specifically, functional correctness of the simulated execution is ensured by executing the synchronization operations in the application faithfully as would be executed on the target parallel machine. The timing correctness of the simulation is accounted for at well-defined points in the application (such as synchronization points). Thus the technique widens the window over which units of simulation work can be executed in parallel without having to synchronize with one another for timing correctness com-

pared to conservative techniques. Further, since the technique never requires having to go back in simulated time (despite the optimism with respect to the timing information) there is no need for rollbacks (and the implied state savings) compared to optimistic techniques. Further we use application specific knowledge (that can be gleaned from the compiler) such as the data layout in the caches to reduce the overhead of simulation. The result is a significant speedup in simulation time for our technique which tracks the speedup inherent in the original parallel application.

The main contributions of this work are:

- a novel technique for parallel simulation of cache-coherent multiprocessors,
- development of performance enhancement strategies aimed at reducing the overheads of parallel simulation,
- a proof of concept prototype implementation that embodies the technique and the enhancements,
- performance results showing the improvement in performance with increasing number of processors.

In Section 2 we present the background and related work on which our research is developed. We next state our assumptions and develop the technique in Section 3. The implementation and related issues are outlined in Section 4. Section 5 gives validation and preliminary results of the implementation. We then identify the overheads in the simulator in Section 6, and suggest implementation ideas and optimizations to reduce these overheads. Using these optimizations, we show the improved performance of the technique for two applications in Section 7. Finally we present some concluding remarks and direction for future research in Section 8.

2 Related Work

Traditionally most cache studies have used either traces or synthetic workloads to drive the simulation. There are drawbacks to both of these approaches. Trace-driven simulation has some validity concerns as observed by several researchers [FH92, GH93, Bit89] due to the distortions that may be introduced due to the instrumentation code that is inserted for collecting the traces. These distortions include non-uniform slowdown of the parallel processes due to varying amount of tracing code in each process, and overall slowdown in the execution speed of all processes owing to tracing. Since the execution path of a parallel program depends on the ordering of the events in the program, both these distortions have the potential of completely changing the execution path unless timing dependencies are carefully eliminated from the traces [GH93]. Program startup effects may also distort the results, especially if the trace length is not long. Further, the traces obtained from

one machine may not represent true interactions in another machine. Lastly, the traces usually do not capture OS related activities (such as interrupts, context switches, and I/O) unless hardware instrumentation is available [SA88]. Many synthetic workload models have been proposed that could avoid the problems associated with traces [AB86, Pop90]. Since application characteristics vary widely, it is difficult to generate a synthetic workload model that is representative of the memory access pattern across a wide range of applications. The drawbacks with both trace-driven and probabilistic simulation can be overcome with an execution-driven simulator. In this approach, real applications are used as the workload on the simulated target architecture. Thus the observed memory access pattern will be the actual one that will be seen on the target architecture. The primary disadvantage of this approach is that it is extremely slow since each instruction has to be simulated for the new architecture in question.

A simple modification to the execution-driven simulation technique has the potential for considerably reducing the simulation time in an execution-driven simulator. In cache simulations, we are interested only in the processor interaction with the cache/memory subsystem. Thus, we simulate only those events that are external to the processor, i.e., those that interact with the memory subsystem and we let the other "compute" instructions execute on the native hardware. In our case, the events of interest include loads, stores and synchronization. This technique of trapping only on "interesting" events saves considerable simulation time and has been used by others [DGH91, BDCW91, CMM+88]. This method, often referred to as "program augmentation", is certainly less expensive than execution-driven simulations in which every instruction is interpreted [Lig92]. An inherent assumption with this method is that instruction fetches do not affect the caching behavior of memory hierarchy. In spite of program augmentation, it may be infeasible (both in terms of space and time) to simulate large system and problem sizes with sequential simulation. Therefore, we explore methods of parallelizing the simulator in this research.

Synchronization of parallel simulators are often characterized as being *conservative* or *optimistic*. For example, in the conservative approach employed in the Wisconsin Wind Tunnel [RHL+92], only events that will not be causally affected by another event are processed in parallel. If t (called the *lookahead*), is the minimum time (for e.g. inter-processor communication time) required for one event to affect another event then we have a range of timestamps (from T to T+t, where T is the current lowest timestamp) that can be processed in parallel. Thus the parallelism is limited to the number of events that fall within this window of size t. Further, deadlocks are a potential problem with some conservative algorithms [CM79]. In the optimistic approach [Jef85], events are processed as soon as they are generated, as though they are independent of others. Such optimism might result in incorrectness in simulating the actual behavior of the application. This situation (when causal violations in processing events are detected) is rectified by rolling back the prematurely executed event computation to a previous correct state, and re-executing the computation to preserve the

causal dependencies. The optimistic approach requires preserving the necessary state information to restart computation in case of a possible rollback. This state saving could be a considerable overhead in simulating the cache behavior of shared memory parallel systems, though incremental state saving may alleviate this problem somewhat.

A parallel cache simulation scheme, based on a time-parallel simulation technique, using program traces has been proposed by Heidelberger and Stone [HS90]. A portion of the program trace is allocated to each processor. In this scheme, each processor assumes an initial state, and simulates its portion of the trace independent of the other processors. The statistics computed by each processor could be wrong due to an incorrect initial state. To address this situation, each processor gets its initial state from its logical predecessor and re-executes the simulation. This step may need to be repeated until the initial state of each processor matches the final state of its logical predecessor. The time-partitioning method is a fairly promising technique since typically there is just one repetition in a cache simulation because of the locality of references. However, it has the usual problem associated with trace-driven methods. In [HS90], it is also shown that it is sufficient to execute the traces for a set of cache lines instead of the entire cache. An implementation of a parallel trace-driven simulation on a MasPar is discussed in [NGLR92], which offers extensions to the above approach.

Dickens et al. [DHN94] suggest a technique for parallel simulation of message-passing programs. Their objective is to simulate the performance of these programs on a larger configuration of a target machine on a smaller host machine. The Wisconsin Wind Tunnel [RHL+92] uses a direct execution approach to simulate a shared memory multiprocessor on a message-passing machine. Using a portion of the memory at each node of CM-5 as a cache, WWT simulates a fine-grained version of shared virtual memory [LH89] through the ECC bits of the CM-5 memory system. The use of ECC bits allows WWT to avoid trapping on each memory operation compared to other execution-driven simulators. Thus only misses and access violations (which manifest as ECC errors in the WWT) are handled through special software trap handlers that simulate the target cache protocol. Using the minimum network latency Q as the *lag*, WWT implements a conservative parallel simulation technique requiring all processors to synchronize every Q cycles for processing the events generated in that window. Even though our goals are similar the approaches are orthogonal (see Section 3) and our technique can further benefit from ideas such as hardware assist to recognize events. Our approach for parallel cache simulation uses a technique that obviates the need for synchronization every Q cycles inherent in conservative approaches such as WWT.

3 The Timepatch Technique

The objective is to develop an execution-driven simulation platform that would enable gathering performance statistics (such as cache hit/miss rates, and message counts) of parallel programs executing on a *target* shared memory parallel machine. The target machine is simulated on a *host* which is also a shared memory parallel machine. Consider a shared memory parallel program with n threads to be executed on a n processor target machine, on which each thread is bound to a unique target processor. We use this program to drive our simulator on the host machine. We map each thread of the program to a separate physical processor of the host machine. Each host processor simulates the activity of the thread (mapped on to it) on the target processor. The simulation has to faithfully model the functional behavior of the original program as well as the timing behavior due to interprocessor communication and synchronization on the target machine. We assume a basic load/store type RISC architecture for the processors of the target machine where interprocessor interactions occur only due to memory reference instructions. We further assume that each target processor has a private cache which is maintained consistent using some cache consistency protocol. Thus the "interesting" events that have to be modeled for gathering the performance statistics of the memory hierarchy of the target machine are the load, store and synchronization operations. Only these interesting events trap into the simulator so that their behavior on the target machine can be modeled faithfully. Upon such traps, the simulator updates the state of the accessed cache block commensurate with the cache protocol implemented on the target machine and performs the intended operation (such as load/store of the data item). The "uninteresting" instructions in the thread are executed at the native speed of the host processor (i.e. they are not simulated at the instruction level). Instead, the time it takes to execute these instructions on the target machine is accounted for in the simulation.

As alluded to above, a simulation has to guarantee two kinds of correctness: *behavioral* and *timing*. A conservative simulation approach addresses both these correctness criteria simultaneously by never allowing event processing to get ahead of the permissible lag. Since no knowledge about the interaction between the events is available to the conservative simulator it must necessarily assume that any event can potentially affect another event outside the lag. In an execution-driven simulation of a shared memory parallel system it is possible to decouple the two correctness criteria as discussed below.

To ensure behavioral correctness it is sufficient if we "simulate" the synchronization events in the parallel program correctly (i.e. as they would happen on the target machine) since these events in turn guarantee the correctness of the shared memory accesses governed by them. The correctness of the shared memory accesses governed by the synchronization events is guaranteed because of the following reason. Our host machine is also a shared memory multiprocessor and thus will

reflect modifications to the shared state affected by the simulated parallel program. Simulating a synchronization event correctly implies ensuring that these events are executed in the same time order as in a conservative simulation, and providing a consistent view of the shared state of the parallel program to all the participating processors at such synchronization points. If the program uses implicit synchronization (i.e. it has data races), then behavioral correctness can still be ensured so long as such accesses can be recognized and flagged by the compiler.

The timing correctness criterion is achieved by a technique called *timepatch* which is described below. Each processor maintains its notion of simulated time which is advanced *locally*. The update of simulated time occurs due to one of two reasons. Firstly, upon executing a block of compute instructions on the host processor a call is made to the simulator to advance the time by the amount of time that block would have taken to execute on the target machine. Secondly, due to the traps into the simulator for the interesting instructions. The load/store events can be to either shared or private memory. These memory accesses may interfere with accesses from other processors of the target machine due to either true or false sharing on the target machine. However, each host processor simulates these load/store accesses as though they are non-interfering with other processors on the target machine and accounts only for the hit/miss timing. The potentially incorrect assumption that these accesses do not interfere with other processors may therefore result in the local notions of simulated time being inaccurate. Thus, at a synchronization point, these timing inconsistencies have to be fixed so that synchronization access is granted in our simulation to the processor that would actually be ahead in time on the target machine and therefore preserve behavioral correctness. Such timing inconsistencies can be corrected as follows. We maintain a history of all the memory accesses on a per processor basis. At synchronization points we merge these history logs to determine the ordering relationship between these accesses. Using this global ordering we can determine the inter-processor interactions (such as invalidation messages) that were previously not accounted and appropriately modify the notion of time of the corresponding processors. Details of how this reconciliation is done are presented in the next section.

The simulation technique outlined above is conservative with respect to the behavioral correctness since it faithfully executes the synchronization events in the parallel program in time order. However, it is optimistic with respect to timing correctness since it allows each processor to account for timing between synchronization points independent of other processors. The price for this optimism is the time overhead in performing the timepatch at synchronization points and the space overhead for recording the history of accesses made by each processor.

Figure 1 demonstrates the timepatch technique with an example. Let us say that the application uses 2 logical processors. We consider only load, store and synchronization operations since those are sufficient to illustrate our idea. Consider the events at time $T_0 2$ on processor 0 and $T_1 2$ on processor 1. As can be seen from Figure 1 there is no order implied between these two events. In

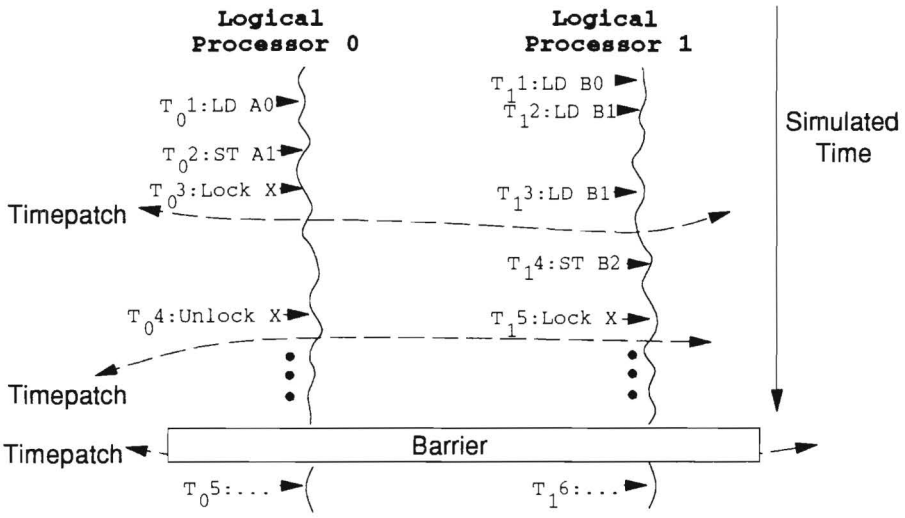

Figure 1: A Timepatch Example

our method, each logical processor is bound to a physical processor and execution is not stalled until synchronization points. Thus in our simulator these events can be processed in parallel. However, if we assume that $A1$ and $B1$ reside on the same cache line and that an invalidation-based protocol is in effect on the target processor, the events $T_0 2$ and $T_1 2$ could indeed affect each other. Assume that $T_0 2$ (store to $A1$) occurs after $T_1 2$ (load of $B1$) but before $T_1 3$ (load of $B1$). In this case, this second load of $B1$ on processor 1 (event $T_1 3$) will be a miss due to the invalidation of that cache line by the store of $A1$ on processor 0 (event $T_0 2$). Observe that even though our hit/miss and timing information may be incorrect, the program's functional correctness is not violated because the shared memory is kept consistent by the underlying host processor. At synchronization points, the timepatch technique has to be applied and timing errors have to be fixed before granting these synchronization requests. Consider the events $T_0 3$ and $T_1 5$ that correspond to the lock requests on processors 0 and 1 respectively. Let us say that on the target processor, processor 0 gets the locks first. However in the simulation it is possible that processor 1 reaches the lock request before processor 0 does due to different speeds of the host processors. To ensure that the behavior on the target machine is accurately reproduced, timepatch is applied at $T_1 5$ and it is determined that processor 0 is lagging behind processor 1 in simulated time and that processor 0 could possibly affect the outcome of this access. Based on this outcome, processor 1 is stalled until the simulated time of other processors (in this case only processor 0) advances to processor 1's current notion of time. Observe that while the timepatch operation is in progress only processor 1 is stalled waiting for other processors to cross time $T_1 5$ so that it can be granted synchronization. The other processors can continue processing the events of their respective threads. Similarly if processor 0 arrived at the event $T_0 3$ first it only needs to stall until processor 1 crosses $T_0 3$. A barrier is a

special synchronization operation and after this operation the simulated time on all the processors participating in the barrier is the same; i.e., values of $T_0 5$ and $T_1 6$ are exactly the same.

The above example illustrates how the timepatch technique works. In the next section, we detail the implementation of this technique on the KSR-2.

4 Implementation

4.1 Description

The target machine to be simulated is a CC-NUMA machine. Each node has a direct-mapped 64 KByte private cache. The shared memory implemented by the target machine is sequentially consistent using a Berkeley style invalidation based cache coherence protocol with a full mapped directory. We assume that local accesses (hits in the simulated cache) cost X cycles while remote accesses (misses in the local cache) cost Y cycles. We also assume that invalidations cost Z cycles irrespective of the amount of sharing. X, Y, and Z thus parameterize the latency attributes of the network that are relevant from the point of view of consistency maintenance.

The above assumption ignores the contention that could result on the network due to remote accesses generated from the processors. As a result the execution time for a parallel application may be worse than what we might observe. Further the performance statistics could also be affected by this assumption. The main motivation for not simulating the network activities in detail is the time overhead for simulating this aspect of the system architecture. The rationale for this assumption is as follows. In a well-balanced design of a parallel architecture one would expect that the network would be able to handle typical loads generated by applications. For example, experimental results on a state-of-the-art machine such as the KSR-2 have shown that the latencies for remote accesses do not vary significantly for a wide variety of network loads [RSRM93]. In [SSRV94], it was reported that the contention overheads observed in several applications were quite small. Recent studies [CKP+93] have also shown that parameterized models of the network may be adequate from the point of view of developing performance-conscious parallel programs. In any event, we do not expect the performance statistics gathered for the memory hierarchy to be affected significantly by ignoring contention since a high hit rate is expected in the private caches of a shared memory multiprocessor implying substantially small amount of network activity.

The host machine is KSR-2 [Res92]. KSR-2 is a COMA machine with a sequentially consistent memory model implemented using an invalidation-based cache coherence protocol. The interconnection network is a hierarchy of rings with 32 processors in the lowest level ring as shown in Figure 2. Each processor node has 512 KBytes of first level cache and 32 MBytes of second level cache. We use a 32-processor KSR-2 for our experiments. We map each node of the target machine on to a unique node on the host machine. The input to the simulator is a parallel program meant to be

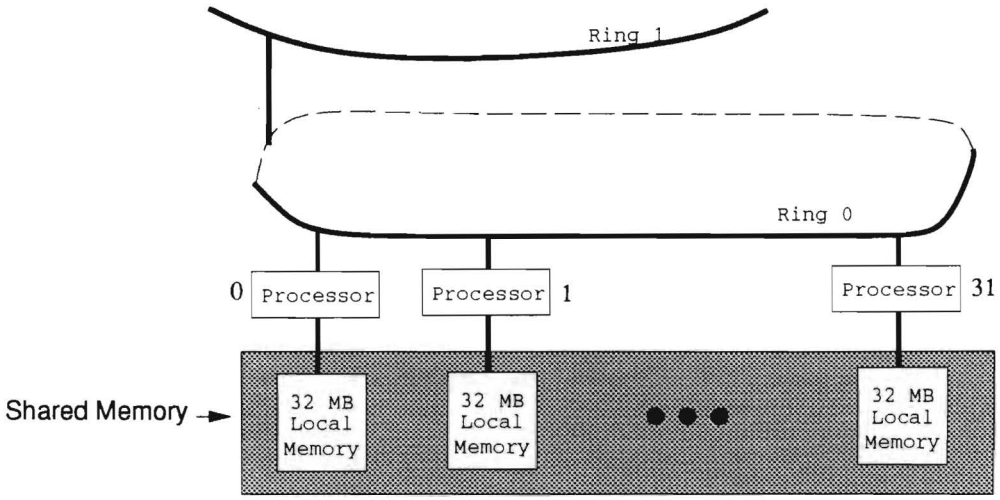

Figure 2: Architecture of the KSR-2

executed on the target machine. As mentioned earlier in Section 2, we use program augmentation technique to execute the "uninteresting" non-memory reference (compute) instructions at the speed of the native host processor. The input program is augmented (currently by hand at the source level) with traps into the simulator for every load, store and synchronization operation.

Typically, cache simulations maintain the directory state of memory, the state of each of the caches and a notion of (global) simulated time. The timepatch approach uses the following information in addition to the above. We maintain a *local* notion of simulated time associated with each processor of the target machine. Events executed on a processor only modify this local time. The timing information is correct in the absence of interactions with other processors. To account for such interactions each processor maintains a table of timestamped events (of all its memory operations). The simulator also maintains a memory directory data structure. Each directory entry in this data structure contains the usual information needed for protocol processing. In addition, the memory directory also maintains a *last served* time which represents the completion timestamp of the last access serviced by the memory directory.

On a trap corresponding to a load/store event, the cache directory state is updated and (hit/miss) time is accounted for based only on the current cache state. Traditional cache simulations also check the state of the memory directory in order to determine if further coherence actions (such as invalidations) are required. We do not perform this step at this point of access. Instead, the processor creates an entry (see Figure 3) in its table which gives the current (local simulated) time, the memory address accessed, the type of operation and an indication if the operation was treated as a hit or a miss. Notice that this operation does not require interaction with any other processor and can proceed independently of other processors. Note also that the possible

Timestamp (Current Time)	Memory Address	Type of Access (Load or Store)	Hit or Miss? (Based on cache state)

Figure 3: An Entry in the Timestamp Table

errors related to timing correctness are that we either treated a memory access as a cache access or that we did not account for the overhead of coherence maintenance. Consequently the local notion of time could be incorrect and possibly less than the "actual" value. For the compute instructions executed on the host processor, the local simulated time is updated based on the time it would have taken on he target processor.

As we mentioned in the previous section, to guarantee behavioral correctness it is necessary to ensure that the synchronization accesses in the parallel program be ordered faithfully in the simulation to correspond to their ordering on the target machine. Thus when a thread reaches a synchronization point in its execution it is necessary to reconcile all the per-processor local notions of simulated times to derive a correct ordering for the synchronization access. This reconciliation involves resolving inter-processor interferences that may have happened due to the load/store memory references that this thread generated up to the synchronization point.

The "timepatch" function (see Figure 4) is called by a processor to perform this reconciliation. A counter $addtime_j$ is associated with each processor j, in which the correction that needs to be applied to the local notion of simulated time is recorded by the timepatch function. Since all memory accesses have to go through the memory directory, it is the central point of conflict resolution for competing memory references from different processors. The timepatch function determines the timing correction that needs to be made to the accesses from each processor based on the interactions between accesses from different processors. This determination is done by applying these accesses to the memory directory when timepatch is called. First the accesses of the processors recorded in the respective timepatch tables have to be ordered before they can be applied to the memory directory. Since each processor timestamps its accesses in order, the local timestamp tables are already sorted in increasing time order. The timepatch routine has to merge all the local timepatch tables, apply them in order to the appropriate memory directory entries, determine the possible interactions among the accesses, and reconcile the ensuing timing inconsistencies. Before we start applying these accesses to the memory directory, we find the timestamp t_j of the last entry in each of the per-processor timestamp tables. We next determine the minimum value Min of the t_j's. Min represents a time up to which all processors have advanced in simulated time. Therefore all the accesses from the processors up to this point of time will be in the timepatch tables, and the potential interaction among these accesses can be resolved. We cannot handle events with timestamps greater than Min at this point because it is possible that the processor

associated with the current minimum Min (the one lagging behind in simulated time) may make accesses that could cause interactions with other accesses that happen in its "future". The first unprocessed entry is picked from each of the local timestamp tables and inserted into a sorted tree if its timestamp is less than Min. Note that there is *at most* one entry per processor at a time in this tree. The record x_j with the minimum timestamp is deleted from the tree and the memory directory entry corresponding to this operation is updated. This update will also determine the increment (if any) that has to be applied to the counter $addtime_j$ of processor j. It should be clear that this increment can never be negative since we process the events in time order. There will be a non-zero increment when the interval between consecutive accesses to the same memory block is less than the servicing time for that access, and represents the queueing delay at the memory directory as well as the cost of consistency maintenance which were overlooked when the access was made. The timestamp of any entry accessed henceforth from the timestamp table corresponding to processor j is incremented by the value in $addtime_j$. If the increment has to be applied to the processor that determined Min originally, then Min will have to be recomputed. The next entry (if any remaining) from this processor's timepatch table is inserted into the tree if it is less than the new Min. The above processing of entries from the tree continues until the tree is empty.

Upon return from the timepatch routine, all inter-processor interferences have been correctly accounted for up to time Min. If the processor that invoked this routine (upon reaching a synchronization point in its execution) still finds itself to be the least in simulated time then it can perform the synchronization access. If not, it would have to block until all the other processors have caught up with it in simulated time to make this determination. It should be noted that while a processor is performing timepatch, other processors can continue with their execution.

We will use the same example used in the previous section (Figure 1) to illustrate in detail how timepatch works. As before, we assume that $A1$ and $B1$ are located on the same cache line. The time order of the events shown in the Figure captures the order in which they should occur on the target machine. Again, we assume that host processor 1 executes faster than host processor 0, and gets to $T_1 5$. Suppose that the only entry in the table associated with processor 0 at this point is $T_0 1$. In the timepatch routine if we process $T_1 3$ then we would not be able to consider the effect of $T_0 2$ on $T_1 3$ since $T_0 2$ has not yet occurred. Thus in this case timepatch should only consider events up to time $T_0 1$. In the meanwhile, processor 1 has to stall until processor 0's time is at least $T_1 5$. Eventually processor reaches $T_0 3$ and timepatch is invoked again because of the synchronization operation. Now both processors 0 and 1 are waiting for the synchronization access. The timepatch routine determines that $T_0 3$ is lesser than $T_1 5$ and grants synchronization access to processor 0. Observe that this is the desired behavior on the target machine.

There are certain subtle cases arising due to the semantics of synchronization events which the implementation has to take care of. Barrier is one such. In this case, the table entries of all the

```
Begin Mutex /* only one processor can perform this action at a given instant */
    (1) for each processor j initialize a counter addtime_j to zero
        /* this keeps track of the correction that needs to be applied
        * to the local notion of that processor's time
        */
    (2) determine t_j, the timestamp of the last entry in each of the
        per-processor tables, and find the minimum Min among these.
        /* timepatch can be performed only up to this time Min */
    (3) pick the first unprocessed event x_j from each per-processor table
    (4) for each x_j if ( x_j.timestamp < Min )
        (4:1) insert x_j into tree sorted by the timestamp x_j.timestamp
        end for /* end of (4) */
    (5) while (tree not empty)
        (5:1) delete a node lt with lowest timestamp from the tree
            (say it belongs to processor j - then lt = x_j )
        (5.2) apply the access of lt to the corresponding memory directory
            and set flag if timing inconsistency detected
            /* this will update the state of the memory directory entry
            * and the timestamp associated with this memory directory
            */
        (5:3) if ( flag is set ) /* implies timing inconsistency */
            (5:3:1) increment the counter addtime_j appropriately
            (5:3:2) Recompute Min
                end if /* end of (5:3) */
        (5:4) pick the next unprocessed entry x_j for processor j
        (5:5) increment x_j.timestamp by addtime_j
        (5:6) if ( x_j.timestamp < Min )
            (5:6:1) insert x_j into the tree
            end if /* end of (5:6) */
            /* continue processing until tree is empty */
        end while /* end of (5) */
End Mutex
```

Figure 4: Pseudo-code invoked to patch time

processors participating in the barrier have to cleared when the last processor arrives at the barrier irrespective of Min. The synchronization traps into the simulator handles such cases correctly.

4.2 Practical Considerations

While timepatching is strictly required only at synchronization points for the correctness of the proposed technique, we are forced to it more often due to space constraints since it is infeasible to maintain very large tables for the timepatch entries. The main idea behind timepatch is to increase the window of time over which it is possible to have parallel execution of the simulated threads of the target architecture. This window gets shrunk a little due to performing timepatch more frequently, but it has the beneficial side effect of advancing the global simulated time, deleting some of the entries from the local timestamp tables, and thus clearing up resources that can be reused.

5 Preliminary Results

The are two aspects to be evaluated to appreciate the merit of our technique. The first is validation of the technique itself to ensure that the performance statistics of the target architecture obtained using our technique is correct. The second is the speedup that is obtained from the parallel simulator. Ideally, we would want to see the speedup curve of the parallel simulator to track the speedup achievable in the original application. However, this depends on how the overheads in the simulation itself gets apportioned among the participating processors. We first address the validation question.

5.1 Validation

We developed a sequential simulator that models the same target machine using CSIM [Sch90], which runs on a SPARC workstation. In order to validate our parallel simulator we used randomly generated traces to drive the sequential and parallel simulators. The performance statistics used to verify the parallel simulator are the simulated cycles (for performing all the memory references in the traces) and the message counts (number of messages generated due to write invalidation). The validation results are shown in Table 1. The traces consist of only load and store references and different proportions of read to write ratios. One of the traces shown has 5000 references and the other has 10000 references. The message counts for the two cases are off by at most 2% for the traces considered. The agreement in simulated times is within 2% for one trace and varies between 7% and 12% for the other trace. CSIM is a process oriented simulation package, which schedules the CSIM-processes in a non-preemptive fashion within a single Unix process. We do not have

Application	Processors	Simulated Cycles			Message Counts		
		Sequential	Parallel	Difference	Sequential	Parallel	Difference
Trace-5K	4	671490	670395	0.16%	6045	6040	0.08%
Trace-5K	8	790010	788600	0.18%	13055	13039	0.12%
Trace-5K	16	868520	883835	-1.76%	27095	27087	0.02%
Trace-10K	4	1519895	1703035	-12.04%	11151	11377	-2.02%
Trace-10K	8	1660345	1781825	-7.32%	27765	27881	-0.41%
Trace-10K	16	1757455	1895990	-7.88%	59980	60010	-0.05%

Table 1: Validation of the Parallel Simulator

any control over the internal scheduling policy that CSIM uses for scheduling the runnable CSIM-processes. We believe that the differences observed for the second trace are due to the effects of this scheduling policy. Overall the validation results indicate that our parallel simulator simulates the target machine with reasonable accuracy.

5.2 Speedup Result

Next we address the speedup question. In order to illustrate and understand the performance of our simulation technique, we will concentrate on the matrix multiplication application. The matrix multiplication problem can be easily parallelized without any false sharing (except for boundary conditions). True sharing is only for read-shared data and thus there is no synchronization in the code. As expected, the raw code we developed for the matrix multiplication application shows linear speedup on KSR-2. Thus we expect that the parallel simulator which uses the same code with augmentation will track the speedup of the application program and give similar speedups. Consider the performance of our simulator when we use 64x64 matrix multiplication to drive our simulation. These results are shown in Table 2. As seen from the Table the speedups observed are not close to what we expect. This is because of the overheads in the simulation. In the next section we analyze what these overheads are and what implementation techniques can be used to reduce these overheads and thus speedup our parallel simulator.

6 The Anatomy of the Parallel Simulator

While performing an execution-driven simulation there are two costs involved - that of the actual application and the simulation overhead (see Figure 5). Let us consider how these costs vary as we simulate different numbers of target processors. For a given problem size, the amount of

Processors	Simulation Time*	Speedup
1	44.27	n-a
2	34.96	1.26
4	31.80	1.39
8	31.23	1.41
16	185.55	0.23

* *Wall clock time in seconds.*

Table 2: Simulation Times for 64x64 Matrix Multiplication

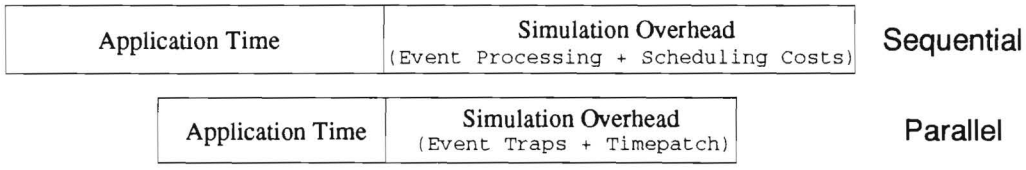

Figure 5: Expected gains using parallel simulation

processor cycles used by the application (assuming a deterministic computation) in a sequential setting is almost a constant since the problem that has to be executed has not changed, irrespective of the number of processors simulated. However, simulating more processors adds to the simulation overhead in the sequential setting because of additional process management and other effects such as decreased cache locality. Now let us consider our parallel simulator. The application time is unaffected by our simulation methodology and hence the available parallelism in the application should be observed in our simulation method as well. The real question is how the simulation overhead is apportioned among the processors when we simulate larger numbers of target processors. The simulation overhead can be broken down into two components: (i) timestamping and local state maintenance on each memory access, and (ii) timepatching at synchronization points. The first component includes the traps incurred for loads and stores. Since these traps and ensuing state maintenance activities are local to each processor, they are done in parallel without interfering with other processors. Therefore assuming that the total number of loads and stores remains a constant for a given problem size the overhead due to the first component also should exhibit the same speedup properties as the application itself, and thus should not limit the speedup of the simulation. The second component of the simulation overhead includes fixing timing errors as well as waits incurred due to local timepatch tables filling up. It is the timepatch function that can be performed only by one processor at a given instant and is the source of the sequential bottleneck in the simulation.

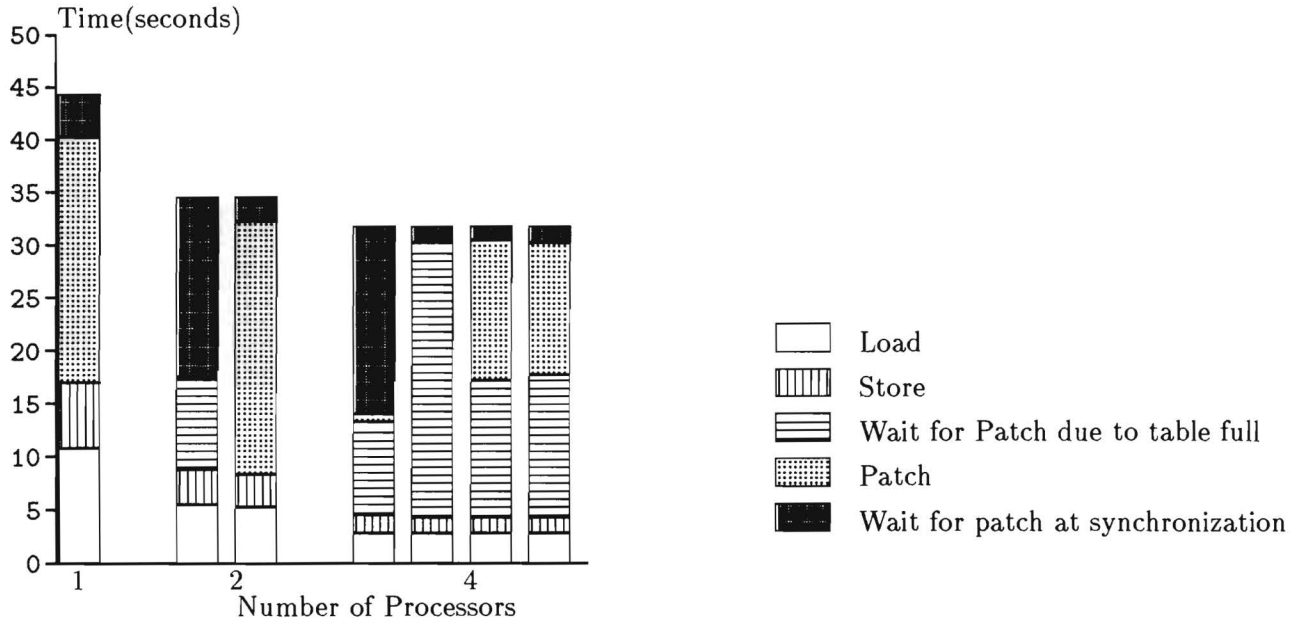

Figure 6: Comparative costs in the 64x64 Matrix Multiplication

To investigate the cause for the relatively poor performance of the simulation we profiled our simulator in order to understand where the time is spent. Figure 6 shows the per-processor breakdown of the various costs involved for different numbers of processors. The application time is itself negligible (less than 1 second for 16 processors). The breakdown gives the time to perform the load traps (labeled *Load* in Figure 6), the store traps (labeled *Store*), the time to perform the timepatch operation (labeled *Patch*), the wait time experienced because the timestamp table is full (*Wait for patch at table full*), and the wait time for the timepatch operation to complete at synchronization (*Wait for patch at synchronization*). The load and store traps are application dependent and are inevitable. The costs associated with performing timepatch and the waits are the simulation overheads of our technique and our aim is to make them as small as possible. From Figure 6, we see that the costs for load/store traps decrease proportionally as we increase the number of processors since this work gets divided among the processors. In the 4 processor case we observe that the wait times due to the timestamp tables becoming full is a significant portion. This preliminary implementation uses a table size of 5K entries per processor. Thus the tables get filled up quite quickly. This forces a processor to attempt to perform a timepatch operation. But if some other processor is already in timepatch then this processor has to wait until some space is freed up in its local table. So reducing the wait time at timepatch is an important step for speeding up the simulation. Another observation from the chart is that the cost to perform the timepatch operation is a significant portion of the total time. Thus another avenue for speeding up the simulation is to speed up the execution of the timepatch operation. The wait time for patch at synchronization is a

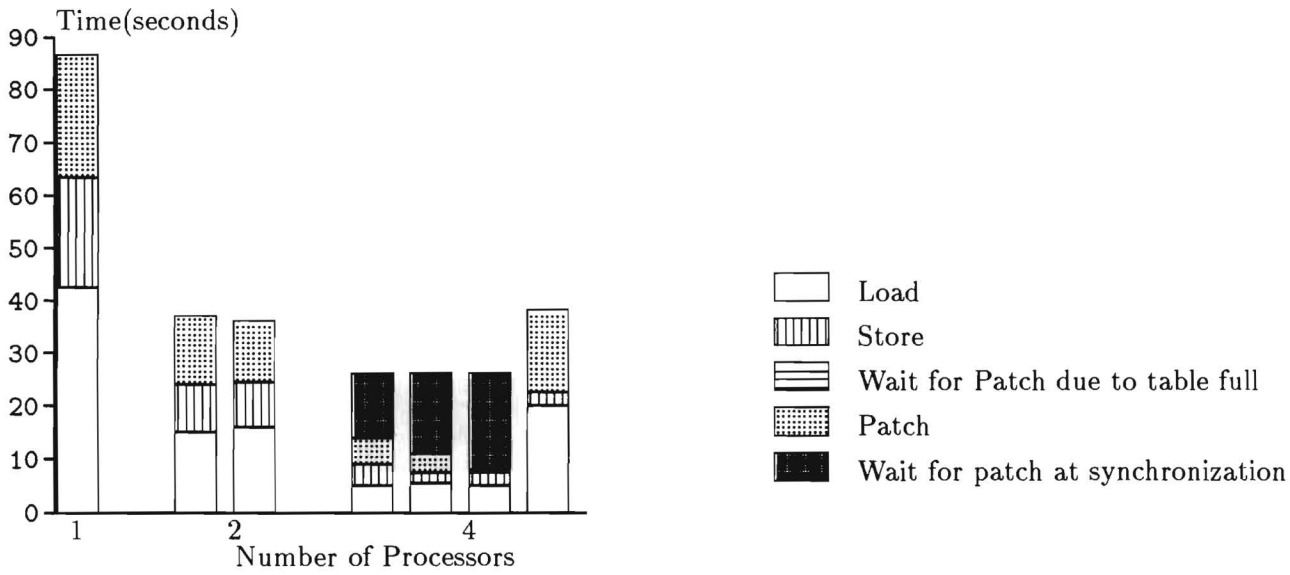

Figure 7: Matrix Multiplication revisited with frequent patching and large table size

subtle issue. If the application has little serial overhead and no work-imbalance, then there should not be any significant waiting at synchronization. The waiting that we do see in Figure 6 is brought about by a processor having to wait at a barrier for some other processor to finish timepatching. In fact, this waiting will go away if we reduce the patch overhead.

In the following subsections, we explore methods to cut down these overheads.

6.1 Reducing the Wait Time at Patch

The processor that is ahead in simulated time will eventually be blocked at a synchronization access. Therefore, one possibility to reduce the wait times at timepatch is to make this processor perform the timepatch operation. Clearly, the total execution time for the simulation cannot be less than the cumulative patch time since timepatch is essentially a sequential process. However, it can be seen from Figure 6 that the total sum of the patch times observed on all the processors is close to the execution time. Therefore, throttling the fastest processor is not going to help. On the other hand, if the size of the timepatch tables are increased then that would reduce the number of trips to timepatch that each processor would have to make and therefore reduce the amount of waiting that each processor would have to do at timepatch. Figure 7 shows the effect of table size on the execution time. As can be seen with a table size of 500K entries, the waiting time at patch is almost close to zero. Also, comparing Figures 6 and 7 it can be seen that increasing the frequency of the timepatch operation did not increase the overall work to be done by the timepatch function.

6.2 Distributing the Patch Work

In our current implementation, the biggest source of simulation overhead is the patching itself. In the current setup this is done sequentially. One possibility to reduce the patch overhead is to parallelize it. A simpler and quicker option is to do the timepatch more often so that it can be overlapped with useful computation on other processors. One way of accomplishing this goal is by performing timepatch periodically instead of waiting until a synchronization operation or until the table is full. In the current implementation timepatch is called every 4096 load/store references. This frequent timepatching in conjunction with the increased table size improves the chance of patch overhead getting distributed among the processors and reduces the possibility of processors waiting at patch when someone else is doing it (see Figure 7).

6.3 Reducing the Patch Time

Given that timepatch is the dominant source of overhead we turn our attention to seeing if that work itself can be reduced. It is instructive to see the number of timepatch entries that are created and the actual number of them that impact the simulated time. For example, in matrix multiply with 16 processors, 798,916 timepatch entries are created but only 61,742 entries (less than 7%) actually result in change to the simulated time due to the inter-processor interferences. It turns out that it is possible for the compiler to predict which references will result in interferences from the data layout the compiler employs for the shared data structures in the caches. Therefore, if this knowledge is available to the simulator then it can eliminate a sizable number of the entries from the timepatch table. We did this optimization for the applications that are considered here. There are three kinds of data: private, true-shared, and false-shared. It is possible for the compiler to distinguish these categories of data, and this information can be made available to the simulator. The simulator should create patch entries for all false-shared data since they could cause interference depending on program dynamics. The simulator does not have to create patch entries for private data since the data layout of the compiler ensures no interference with other processors. The true shared data gets shared across synchronization regions in the program. Thus within a synchronization region it is sufficient if there is at least one patch entry for the first access for each such variable in that region by a processor.

We modified our implementation to use these optimizations. Associated with each cache block is a timestamp that gives the last access to that cache block. We associate a *sync-time* with each processor. This is the timestamp of the last synchronization operation carried out by this processor. The augmentor which generates the load/store traps gives this additional information whether the access is to private, false-shared, or true-share data. Upon every true shared access the timestamp in the referenced cache block is checked against the sync-time for that processor. If the sync-time

Processors	Unoptimized time (Secs)	Optimized time (Secs)	Improvement over Unoptimized	Speedup (for optimized)
1	68.20	39.28	42%	1
2	35.22	17.82	49%	2.2
4	26.65	13.85	48%	2.8
8	29.19	10.61	63%	3.7
16	160.15	56.16	65%	0.6

Table 3: Comparing the optimized and unoptimized Matrix Multiplication Simulation

Processors	Unoptimized time (Secs)	Optimized time (Secs)	Improvement over Unoptimized	Speedup (for optimized)
1	164.20	94.01	42%	n.a
2	131.18	45.53	65%	2.1
4	131.35	33.41	74%	2.8
8	241.61	52.55	78%	1.7
16	1301.11	398.21	69%	0.2

Table 4: Comparing the optimized and unoptimized Integer Sort Simulation

is greater then an entry is created for this reference in the patch table. An entry is created for every false-shared access and none for private.

With these optimizations the number of timepatch entries that are actually processed during the entire simulation is down from 798,916 for 16-processor matrix multiply to 267,636 (by a factor of 3). This is still around a factor 4 more than the number of entries that actually cause timing errors. By a more careful analysis and labeling of the accesses we expect to bring down this number even further.

The cache statistics (such as message counts and hit rate) observed for the optimized and unoptimized versions were the same indicating that these optimizations are indeed correct. Since the objective here is to show the viability of our simulation technique these performance statistics are not germane to the rest of the discussion. In the next section we present the speedup results from the optimized simulator for two application programs: matrix multiply, and integer sort.

7 Results from the Optimized Simulator

Tables 3, and 4 show the execution times and speedups for matrix multiply and integer sort.

Application	Processor id	User Time			Wall Clock Time		
		Load+Store	Wait	Patch	Load+Store	Wait	Patch
16-processor Matrix Multiplication	Processor 0	1.51	0.47	7.653	1.82	15.26	37.39
	Processors 1-15	1.49	1.55	0.002	1.96	53.30	0.05
8-processor Integer Sort	Processor 0	10.31	1.55	4.06	18.21	14.86	48.88
	Processors 1-8	9.41	1.78	2.54	11.57	24.82	3.43

Table 5: Breakdown of the user and wall clock times

As can be seen, we do get a reasonable speedup up to 8 processors for matrix multiply (3.7 for 8 processors), and up to 4 processors for integer sort (2.8 for 4 processors). Comparing the execution times of the optimized and unoptimized versions of the simulator we can see that the optimizations give a significant payoff (ranging from 42% to 78%). Since both versions use the same table sizes and periodicity for timepatch, the reduction in time is entirely due to the compile time optimizations which we discussed in Section 6.3.

Now let us try to understand the reason for slowdown beyond 8 processors for matrix multiply and 4 processors for integer sort. Table 5 shows the breakdown of the execution times for 16 processor matrix multiply, and 8 processor integer sort. The column labeled *user time* is the time spent in the simulator; the *wall clock time* is the elapsed time seen for the simulator. As can be seen one processor (labeled 0 in the Table) does most of the patching in matrix multiplication. The cumulative user time spent in patching was observed to be commensurate with that for lesser numbers of processors. However, it is seen that the wall clock time is greater than this patch time by 29.74 Secs. This is additional time spent by processors 1-15 waiting at the synchronization point for the patch function on processor 0 to complete. We believe we know exactly why this behavior is observed. KSR-2 has 32 MBytes of second level cache, and it allocates space for an entire page (of 16 KBytes) on every second level cache miss. Each per-processor patch table is a 2 Mbyte data structure. We believe that with 16 processors, the necessity to consult the tables of other processors at patch time results in a considerable amount thrashing of the second level cache leading to poor overall performance. This is the same effect which is observed in integer sort beyond 8 processors. Needless to say, that this behavior is purely an implementation quirk and does not impact the usefulness of the timepatch technique. In fact, we expect to be able to fix this interaction between the simulation data structures and the paging behavior of the operating systems by careful data partitioning.

8 Concluding Remarks

We developed a new method for parallel simulation of multiprocessor caches and have shown its feasibility by implementing it on the KSR-2. The primary advantages of this scheme are that we can obtain reasonable speedups limited primarily by the application speedups and that this method can be used to simulate larger parallel systems (both number of target processors and problem size) than is possible with a sequential simulator.

In the process of implementing the technique on KSR-2 we learned several lessons which are of interest from the point of view of performance tuning parallel applications in general, parallel simulators in particular. The first has to do with the potential for considerable speedup for parallel simulation of multiprocessor caches by gleaning compile time information on the data layout in the caches and passing it on to the simulator. The second is the importance of distributing the overhead of the simulation among the processors so that useful work in other processors can be overlapped with this overhead function. The last is the necessity to worry about the operating system interactions in a parallel machine such as the KSR-2. In particular, one has to carefully orchestrate the interactions between the shared data structures in the application and the paging policies of the operating system. In fact, this last point is not resolved fully in our current implementation. But we are sure that we can get it resolved and show considerable speedups for larger numbers of processors.

There is at least one architectural implication suggested by this parallel simulation exercise. It is clear that the performance of our technique can benefit from a "snoopy-read" primitive. The semantics of this primitive is to read the current value of a variable without changing the (exclusive) state of the cache line on the processor from which the value is being read. The timepatch routine could make heavy use of this feature. In the absence of such a primitive, the referenced cache line thrashes between the processors and not only slows down the execution on the affected processors but also places considerable stress on the interconnection network.

There are several directions to extend our work. One direction is to figure out a way to incorporate network contention of the target architecture into the simulator. This direction would allow extending our technique to simulate both memory and I/O intensive applications (which may stress the network) in addition to the compute intensive ones we have studied so far. A possible approach to incorporate the network would be to combine our method with an optimistic scheme such as Time warp to simulate the network messages.

A second direction is to use this technique to compare different memory systems employing different models of consistency and cache coherence strategies. A third direction is to simulate larger configuration of target architectures on smaller host machines. In our implementation, we used one host processor to simulate one target processor. The timepatch technique does not place

any such constraints and we could just as easily have mapped multiple nodes of the target machine to a single processor of the host machine. We also assumed that each target processor has only one thread of the application mapped to it. To relax this restriction we have to take into account the scheduling strategy on the target machine so that the cache effect produced by multiple threads on a processor can be accurately modeled.

References

[AB86] J. Archibald and J. L. Baer. Cache coherence protocols: Evaluation using a multiprocessor simulation model. *ACM Transactions on Computer Systems*, 4(4):273–98, November 1986.

[ASHH88] A. Agarwal, R. Simoni, J. Hennessy, and M. Horowitz. An evaluation of directory schemes for cache coherence. In *15th Annual International Symposium on Computer Architecture*, pages 280–9, May 1988.

[BDCW91] Eric A. Brewer, Chrysanthos N. Dellarocas, Adrian Colbrook, and William E. Weihl. PROTEUS: A high-performance parallel-architecture simulator. Technical Report MIT/LCS/TR-516, Massachusetts Institute of Technology, September 1991.

[Bit89] Philip Bitar. A critique of trace-driven simulation for shared-memory multiprocessors. In *16th ISCA Workshop Presentation*, May 1989.

[CKP+93] D. Culler, R. Karp, D. Patterson, A. Sahay, K. E. Schauser, E. Santos, R. Subramonian, and T. v. Eicken. Logp: towards a realistic model of parallel computation. In *Fourth ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, pages 1–12, July 1993.

[CM79] K. M. Chandy and J. Misra. Distributed simulation: A case study in design and verification of distributed programs. *IEEE Transactions on Software Engineering*, SE-5(5):440–452, September 1979.

[CMM+88] R. C. Covington, S. Madala, V. Mehta, J. R. Jump, and J. B. Sinclair. The Rice parallel processing testbed. *Performance Evaluation Review*, 16(1):4–11, May 1988.

[DGH91] Helen Davis, Stephen R. Goldschmidt, and John Hennessy. Multiprocessor simulation and tracing using TANGO. In *International Conference on Parallel Processing*, pages II–99–107, 1991.

[DHN94] P. M. Dickens, P. Heidelberger, and D. Nicol. A distributed memory LAPSE: Parallel simulation of message passing programs. In *8th Workshop on Parallel and Distributed Simulation*, pages 32–38, July 1994.

[EK88] S. J. Eggers and R. H. Katz. A characterization of sharing in parallel programs and its application to coherency protocol evaluation. In *15th Annual International Symposium on Computer Architecture*, pages 373–82, June 1988.

[FH92] Richard M. Fujimoto and William C. Hare. On the accuracy of multiprocessor tracing techniques. Technical Report GIT-CC-92-53, Georgia Institute of Technology, November 1992.

[Fuj83] R. M. Fujimoto. Simon: A simulator of multicomputer networks. Technical Report UCB/CSD 83/137, ERL, University of California, Berkeley, 1983.

[Fuj90] Richard M. Fujimoto. Parallel discrete event simulation. *Communications of the ACM*, 33(10):30–53, October 1990.

[GH93] S. R. Goldschmidt and J. L. Hennessy. The accuracy of trace-drive simulations of multiprocessors. In *ACM Sigmetrics Conference on Measurement and Modeling of Computer Systems*, pages 146–57, June 1993.

[HS90] Philip Heidelberger and Harold S. Stone. Parallel trace-driven cache simulation by time partitioning. In *Winter Simulation Conference*, pages 734–737, December 1990.

[Jef85] D. R. Jefferson. Virtual time. *ACM Transactions on Programming Languages and Systems*, 7(3):404–425, July 1985.

[LH89] Kai Li and Paul Hudak. Memory coherence in shared virtual memory systems. *ACM Transactions on Computer Systems*, 7(4):321–359, November 1989.

[Lig92] Walt B. Ligon. *An empirical analysis of Reconfigurable Architectures.* PhD thesis, Georgia Institute of Technology, August 1992.

[NGLR92] David M. Nicol, Albert G. Greenberg, Boris D. Lubachevsky, and Subhas Roy. Massively parallel algorithms for trace-driven cache simulation. In *6th Workshop on Parallel and Distributed Simulation*, pages 3–11, January 1992.

[Pop90] D. A. Poplawski. Synthetic models of distributed memory parallel programs. Technical Report ORNL/TM-11634, Michigan Technological University, September 1990.

[Res92] Kendall Square Research. Technical summary, 1992.

[RHL+92] Steven K. Reinhardt, Mark D. Hill, James R. Larus, Alvin R. Lebeck, James C. Lewis, and David A. Wood. The Wisconsin Wind Tunnel: Virtual prototyping of parallel computers. Technical report, University of Wisconsin-Madison, November 1992.

[RSRM93] U. Ramachandran, G. Shah, S. Ravikumar, and J. Muthukumarasamy. Scalability study of the ksr-1. In *International Conference on Parallel Processing*, pages I–237–240, August 1993.

[SA88] R. L. Sites and A. Agarwal. Multiprocessor cache analysis using ATUM. In *15th Annual International Symposium on Computer Architecture*, pages 186–95, June 1988.

[Sch90] Herb D. Schwetman. CSIM Reference Manual (Revision 14). Technical Report ACA-ST-252-87, Microelectronics and Computer Technology Corp., Austin, TX, 1990.

[SSRV94] A. Sivasubramaniam, A. Singla, U. Ramachandran, and H. Venkateswaran. An approach to scalability study of shared memory parallel systems. In *ACM SIGMETRICS Conference on Measurement and Modeling of Computer Systems*, pages 171–180, May 1994.